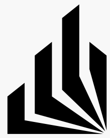

Universität
Gesamthochschule Paderborn

Fachbereich 17 • Mathematik – Informatik

Cliché - Erkennung in relationalen Datenbankanwendungen

DIPLOMARBEIT

(Studiengang Wirtschaftsinformatik H II)

von

Barbara Bewermeyer
Thunemühle 60
33104 Paderborn
Matrikel-Nr.: 3059078

vorgelegt bei

Prof. Dr. W. Schäfer

und

Prof. Dr. G. Engels

im

September 1998

Danksagung

Diese Arbeit wurde am Lehrstuhl Softwaretechnik der Universität-Gesamthochschule Paderborn erstellt. Mein Dank gilt Herrn Dipl. Inf. Jens Jahnke für die interessante Aufgabenstellung und seine wertvollen Anregungen und Hinweise, sowie Herrn Prof. Dr. W. Schäfer für die begleitende Betreuung und Herrn Prof. Dr. G. Engels für die Zweitkorrektur der Arbeit. Außerdem danke ich allen, die mir im Verlauf dieser Arbeit zur Seite standen.

Eidesstattliche Erklärung

Ich versichere, die vorliegende Diplomarbeit selbständig angefertigt, alle benutzten Hilfsmittel vollständig angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer direkt oder mit Abänderungen entnommen wurde.

Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Paderborn, den 4. September 1998

(Barbara Bewermeyer)

Einleitung 1
Motivation 1
Kontext der Arbeit 3
Ziel der Arbeit 5
Aufbau der Arbeit 6
Grundlagen relationaler Datenbanken 7
Phasenmodell des Datenbankentwurfs 7
Beispielszenario 10
Konzeptioneller Entwurf 10
Relationaler Datenbankentwurf 13
Optimierungen beim physischen Datenbankentwurf 18
Die relationale Datenbanksprache SQL 19
Reverse Engineering 23
Ansatzpunkte für das Reverse Engineering 23
1. Ansatzpunkt: Datenbankschema 25
2. Ansatzpunkt: Datenbankausprägungen 26
3. Ansatzpunkt: Datenbankzugriffe 28
Probleme des Reverse Engineerings 30
Cliché-Erkennung 31
Software-Clichés 31
Möglichkeiten zur Erkennung von Clichés 32
Textparsing 34
Graphparsing 39
Der Recognizer von Linda Wills 39
Der Hybridansatz von Alex Quilici 44
Clichés im Bereich relationaler Datenbanken 46
Cliché-Erkennung in relationalen Datenbankanwendungen 47
Konzeption und Realisierung des Clichéparsers 49
Konzeption 49
Geschichtete Graphgrammatiken 54
Realisierung des Clichéparsers 57
Unterschiede zwischen Konzeption und Realisierung 60
Erweiterung der Clichébibliothek 65
Spezifikation der Clichés 67
Clichés zur Erkennung von intrarelationalen Abhängigkeiten 67
Select-Distinct Cliché 67
Cyclic-Exclusion Cliché 72
Group-By Cliché 75
Complex Cliché 77
Clichés zur Erkennung von interrelationalen Abhängigkeiten 79
Join Cliché 79
Clichés zur Erkennung von Optimierungsstrukturen 83
Many-to-Many Cliché 83

Many-to-Many-Select-Distinct Cliché 86
Zusammenfassung und Ausblick 89
Anhang 91
SQL-DML Grammatik 91
Literatur 97

1 Einleitung

1.1 Motivation

„Die gesamte Zeitspanne von der Projektbegründung über die Einführung bis zur späteren Ablösung des DV-Anwendungssystems durch ein völlig neues wird [...] als *Software-Lebenszyklus* (software life cycle) bezeichnet. Da dieser Zyklus nach Erfahrungswerten 15 Jahre und länger dauern kann, ist für die Wartung von DV-Anwendungssystemen oft ein Mehrfaches des Entwicklungsaufwands aufzubringen.“ [STA89]

Nach [STA89] versteht man unter der Wartung von bereits im Einsatz befindlichen Programmen ihre Anpassung an Änderungswünsche des Anwenders bzw. an geänderte (gesetzliche, organisatorische oder sonstige) Rahmenbedingungen. Diese Anpassungen machen es oftmals erforderlich, die ursprüngliche Konzeption einer Anwendung zu ändern. Es ist daher unabdingbar, diese Konzepte und ihre Verwirklichung genau zu beschreiben und jede Änderung zu dokumentieren, um einen reibungslosen Wartungsbetrieb zu ermöglichen.

Die vorhandene Struktur der DV-Anwendungen zu kennen, ist aber nicht nur notwendig für deren Wartung, sondern insbesondere auch für die Analyse der bisherigen betrieblichen Abläufe bei der Umstrukturierung und Neugestaltung, dem *Reengineering*, von Informationssystemen. Nur wenn man die bisherige Vorgehensweise kennt und versteht, ist es möglich, Schwachpunkte herauszufinden und Verbesserungen zu erarbeiten, die dann in neugestalteten DV-Systemen berücksichtigt werden. Ziel heutiger Informationssysteme ist die Integration der verschiedensten betrieblichen Bereiche zu einer Einheit, die es erlaubt, spezielle Anforderungen einzelner Bereiche zu berücksichtigen, aber trotzdem eine einfache Handhabung der Systeme ermöglicht. So ist z.B. die Bereitstellung der Daten in einem vom Anwender benötigten, speziellen Format vorzunehmen, aber dennoch eine unternehmensweite Integration der Daten anzustreben, wie es bei Informationssystemen des Data Warehousing versucht wird. Gerade auch im Zuge der überall notwendigen Umstellungen der bisher eingesetzten Informationssysteme an die Anforderungen des nächsten Jahrtausends sollte eine Aktualisierung der Dokumentation und eventuell eine Neukonzeption veralteter Systeme in Angriff genommen werden.

Dabei ist auch eine Analyse der benutzen Datenstrukturen im Hinblick auf die Eignung für zukünftige Informationssysteme von besonderer Wichtigkeit, da neue Anwendungs-

bereiche z.B. im Computer-Aided Design (CAD), Computer-Integrated Manufacturing (CIM) oder Software Engineering (SE) andere Anforderungen an die Informationsverarbeitung stellen als traditionellen Anwendungen z.B. aus dem Rechnungswesen. So ist die Verwaltung komplexer Strukturen, wie sie in den neuen Anwendungsbereichen typischerweise genutzt werden, problematisch. Relationale Datenbanken, die üblicherweise für die persistente Speicherung der betrieblichen Daten genutzt werden, wurden zur Verwaltung von Daten einfacher Struktur entwickelt. Hierarchische Strukturen lassen sich hierauf nicht adäquat abbilden. Durch die Abbildung dieser komplexen Strukturen auf das relativ einfache Relationenmodell, das relationalen Datenbanken zugrundeliegt, gehen semantische Informationen verloren, so daß schon allein hierdurch eine Abweichung der Dokumentation von der tatsächlichen Implementierung entstehen kann.

Aber auch wenn bei der Entwicklung der Informationssysteme eine vollständige und korrekte Dokumentation angefertigt wurde, wird die Notwendigkeit einer vollständigen und aktuellen Dokumentation für laufende Anwendungen oft nicht beachtet. Hier ist es vielmehr die Regel, daß anfallende Änderungen an Programmen, die sich teilweise bereits jahrzehntelang im Einsatz befinden, nur noch konzeptionslos durchgeführt werden und eine Anpassung der Dokumentation ganz ausbleibt. In vielen Unternehmen ist demnach für die genutzten Informationssysteme kaum bzw. nur veraltete Dokumentation vorhanden und auch die ursprünglichen Entwickler können oft nicht mehr befragt werden. Eine Folge hiervon ist, daß unstrukturierte, schlecht überschaubare Anwendungen (sogenannte *Legacy-Systeme*) entstehen. Es stehen keine detaillierten Informationen über den genauen Aufbau und die Architektur der einzelnen Anwendungen zur Verfügung und gegebenenfalls muß im Quelltext der Ablauf eines Programmes mühsam nachvollzogen werden. Diese manuelle Analyse gestaltet sich aber schwierig oder ist gar unmöglich, da Programme, die über die Jahre „gewachsen“ sind, oft mehrere 100 000 Codezeilen umfassen. Der Quelltext ist aber die einzige Komponente der Informationssysteme, in der im Laufe der Jahre die Änderungen vorgenommen wurden. Er ist somit die einzige Quelle, aus der aktuelle Informationen zu bekommen sind.

Mit dem Problem der Redokumentation und Analyse vorhandener Informationssysteme beschäftigt sich das *Reverse Engineering*. Dabei ist zu berücksichtigen, daß betriebliche Informationssysteme zu einem großen Teil datenorientiert sind, d.h. sie befassen sich zumeist mit der Verwaltung und Aktualisierung aller Informationen, die das Unternehmen für die Erledigung der täglichen Aufgaben benötigt. Eine persistente Speicherung dieser betrieblichen Daten erfolgt üblicherweise in relationalen Datenbanken. In diesen datenorientierten Systemen kann die Struktur der persistenten Daten getrennt von den hierauf zugreifenden Anwendungen, die teilweise komplexe Abläufe beschreiben, durchgeführt werden, um so den Reverse Engineering-Prozeß zu vereinfachen. Zudem ist die Struktur der persistenten Daten oftmals die Komponente eines Informationssystems, die den wenigsten Änderungen unterworfen wurde. Ein Reverse

Engineering der vorhandenen Informationssysteme sollte deshalb hier ansetzen und zunächst die statischen Strukturen der zugrundeliegenden Datenbank(en) analysieren. Eine solche Vorgehensweise bezeichnet man als *Datenbank Reverse Engineering* (DBRE) [HEH+98]. Zur Analyse der Datenbankstrukturen gibt es bereits einige Ansätze, die sich aber oft auf die Analyse des physischen Schemas beschränken (vgl. [FV95], [PB95]). Allein hieraus lassen sich aber nicht alle Informationen gewinnen, um ein detailliertes konzeptionelles Schema abzuleiten. Erst eine Analyse der vorhandenen Daten und vor allem der Datenbankzugriffe ermöglicht dies (vgl. [AND94], [HEH+98], [JSZ97]). Oft ist auch die Einbeziehung von Expertenwissen unumgänglich. Mit der Untersuchung und der semantischen Analyse der Datenbankzugriffe befaßt sich diese Arbeit. Es gibt bereits einige Ansätze (vgl. [PP94], [QUI94], [RW90]), die nur anhand des Quelltextes versuchen, automatisch das Design eines Programmes nachzuvollziehen. Allerdings befassen sich diese nicht speziell mit der Problematik der Datenbankzugriffe. In den genannten Ansätzen werden die Programme nach typischen Mustern (sog. *Clichés*), deren Semantik bekannt ist, durchsucht, so daß Aussagen über benutzte Datenstrukturen oder typische Algorithmen wie z.B. Sortierverfahren möglich sind. In dieser Arbeit wird versucht, die Erkenntnisse und Vorgehensweisen dieser allgemeinen Ansätze zu nutzen und sie speziell an die Erfordernisse von Datenbankanwendungen anzupassen.

1.2 Kontext der Arbeit

Der folgende Abschnitt beschreibt eine Umgebung für das Reengineering von relationalen Datenbanken, wie sie im Projekt VARLET (Verified Analysis and Reengineering of Legacy Database Systems Using Equivalence Transformation) an der Universität Gesamthochschule Paderborn entwickelt wird. In diesem Rahmen entstand auch die vorliegende Diplomarbeit.

Ziel dieses Projektes ist es, eine Umgebung anzubieten, die es dem Anwender ermöglicht, eine Analyse der oben beschriebenen, über die Jahre gewachsenen Datenbanksysteme (Legacy-Systeme) durchzuführen und das relationale Schema in ein konzeptionelles objektorientiertes umzuwandeln. Den Migrationsprozeß schließt eine Überführung der Daten aus der relationalen Datenbank in eine objektorientierte Middleware ab, so daß es möglich wird, neue objektorientierte Anwendungen zu entwickeln, ohne gleichzeitig auf die gewohnte und stabile Datenhaltung in den vorhandenen relationalen Datenbanken verzichten zu müssen. Auf diese Weise ist es natürlich auch möglich, autonome Legacy-Anwendungen weiter zu nutzen, die wie bisher direkt auf die relationale Datenbank zugreifen können bzw. eine schrittweise Anpassung aller Anwendungen vorzunehmen.

Wie oben bereits beschrieben, ist es für das Reengineering von Informationssystemen unerläßlich, eine vollständige und aktuelle Dokumentation der bisherigen Anwendungen bzw. zumindest der benutzten Datenstrukturen zu kennen. Diese ist aber häufig nicht vorhanden. Der erste Schritt eines Migrationsprozesses ist somit die Analyse der Legacy-Systeme. Bei Datenbanksystemen können semantische Informationen aus dem vorhanden Schema, den Ausprägungen der Daten in der Datenbank und den Anwendungen, die auf die Datenbank zugreifen, gewonnen werden [JSZ97]. Hieraus läßt sich ein semantisch angereichertes Datenbankschema ableiten, das dann in ein erstes äquivalentes objektorientiertes Schema transformiert wird. Sowohl der Analyse- als auch der Transformationsschritt kann nur in Interaktion mit dem Nutzer vollzogen werden, da es denkbar ist, aus einem relationalen Schema verschiedene objektorientierte Schemata abzuleiten. Des weiteren werden der Analyse- und der Transformationsschritt inkrementell und iterativ durchgeführt, d.h. es können dem Reengineeringprozeß immer wieder neue Informationen hinzugefügt werden, die dann nachträglich noch Berücksichtigung finden (vgl. [JSZ97]). Die Äquivalenz zwischen den beiden Schemata wird durch eine entsprechende Abbildung garantiert. Das objektorientierte Schema, das aus dieser inkrementellen Transformation entsteht, berücksichtigt dann alle semantischen Informationen des relationalen Systems und nutzt zudem, so weit dies möglich ist, objektorientierte Modellierungskonzepte. Es kann dann in einem weiteren Schritt vom Nutzer mit Hilfe eines Redesigntools angepaßt werden. Dieses Tool stellt dann wiederum über die Abbildungsinformationen eine Äquivalenz zum relationalen Schema sicher.

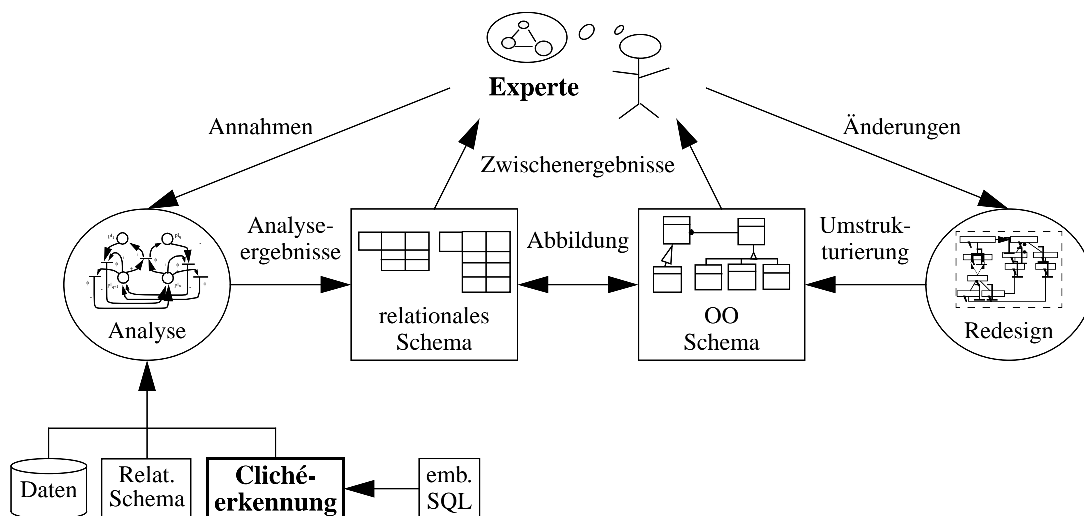


Abbildung 1.1 *VARLET-Reengineering-Umgebung*

Die Interaktion mit dem Anwender ist während des gesamten Migrationsprozesses ein wichtiger Faktor, um ein optimales Ergebnis zu erhalten. Hierzu stellen sowohl das Analyse- als auch das Redesigntool Editoren zur Verfügung, die die relationalen bzw. objektorientierten Schemata im UML-Format (Unified Modelling Language) darstellen

und die Möglichkeit bieten, dem entsprechenden Prozeß weitere Informationen (Expertenwissen) zuzuführen.

1.3 Ziel der Arbeit

Die Überführung eines relationalen in ein objektorientiertes Datenbanksystem erfordert zuallererst die genaue Kenntnis des konzeptionellen Schemas der relationalen Datenbank. Dieses Schema ist Voraussetzung für die Ableitung eines äquivalenten objektorientierten Schemas.

Da das physikalische Schema viele, für die Migration wichtige Informationen nicht explizit enthält, werden auch die Ausprägungen der Daten in der Datenbank und die relationalen Datenbankanwendungen untersucht. Hiermit beschäftigt sich diese Arbeit. Sie versucht semantische Informationen aus den Anwendungen herauszufiltern, die häufig implizit im Code versteckt sind. Beispiele hierfür sind referentielle Integritätsbedingungen, Schlüsselkandidaten, sowie Vererbungs- und Aggregationsstrukturen. Hinweise auf solche Strukturen können gewonnen werden, indem die Datenbankanfragen nach bestimmten Mustern durchsucht werden. Wie oben bereits erwähnt, werden solche typischen Muster im Bereich des Reengineering auch als Clichés bezeichnet.

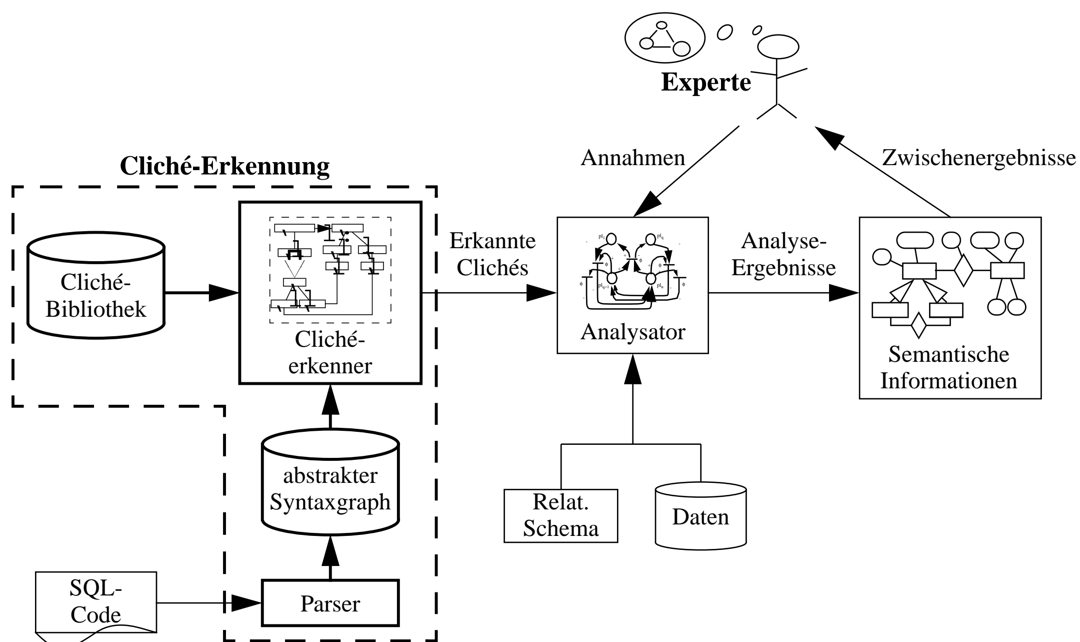


Abbildung 1.2 *Cliché-Erkennung in Varlet*

Im Rahmen dieser Arbeit werden für den Bereich der semantischen Analyse von Datenbanken typische Clichés motiviert, formal anhand von Graphgrammatiken definiert und eine Möglichkeit zur automatischen Erkennung dieser Clichés aufgezeigt.

1.4 Aufbau der Arbeit

Im folgenden Kapitel wird zunächst ein Beispielszenario vorgestellt, anhand dessen dann die Grundlagen von relationalen Datenbanken und hier insbesondere das Vorgehen beim relationalen Datenbankentwurf erläutert werden.

In Kapitel 3 wird aufgezeigt, wie aus einem vorhandenen Datenbanksystem nur durch Analyse des Schemas, der Datenbankanwendungen und der Ausprägungen der Daten ein konzeptionelles Schema abgeleitet werden kann.

In Kapitel 4 wird dann das Reverse Engineering der Datenbankanwendungen (SQL-Code) durch Erkennung von typischen Mustern (Clichés) im Code näher untersucht. Dabei werden nicht nur Möglichkeiten aufgezeigt, Clichés zu erkennen, sondern auch auf Besonderheiten aufmerksam gemacht, die Clichés im Bereich relationaler Datenbanken auszeichnen.

Das Kapitel 5 beschäftigt sich mit der Konzeption und Realisierung des Programmoduls „Clichéparser“, das die automatische Erkennung der Clichés durchführt.

Eine Spezifikation der in dieser Arbeit betrachteten Clichés folgt dann in Kapitel 6.

Zuletzt faßt Kapitel 7 die Ergebnisse dieser Arbeit zusammen und gibt einen Ausblick.

2 Grundlagen relationaler Datenbanken

In der Einleitung wurde bereits erwähnt, daß die persistente Struktur einer Datenbankanwendung zumeist die stabilste Komponente der betrieblichen Informationssysteme darstellt [HEH+98]. Die Informationen, die beim Entwurf dieser Strukturen gesammelt und zur Analyse benutzt wurden, gehen häufig aber mit der Zeit verloren bzw. sind teilweise gar nicht dokumentiert. Auch an der Datenbank vorgenommene Änderungen und Anpassungen an neue Anforderungen führen dazu, daß die während der Planung der Legacy-Systeme entstandenen Dokumente im Laufe der Jahre veralten und ungültig werden, wenn sie nicht an die Änderungen angepaßt werden. In diesem Kapitel soll aus diesem Grunde zunächst ein Überblick über den Entwurf von Datenbanken gegeben werden. Denn nur, wenn man sich darüber klar ist, welche Informationen in welchen Phasen des Datenbankentwurfes benötigt werden und teilweise im Laufe des Entwurfsprozesses wieder verloren gehen, kann man beim Reverse Engineering von Legacy-Systemen diese fehlenden Informationen wieder herstellen. Anhand eines Beispielszenarios wird dann genauer auf den konzeptionellen, logischen und physischen Entwurf eingegangen.

Hier kann leider nicht auf alle Aspekte dieser Thematik detailliert eingegangen werden. Es soll lediglich ein Überblick gegeben werden. Nähere Informationen sind z.B. [EN94] oder [HS95] zu entnehmen.

2.1 Phasenmodell des Datenbankentwurfs

Den Datenbankentwurf kann man als Spezialisierung des allgemeinen Softwareentwurfs ansehen, da man auch hier nach einem Phasenmodell vorgeht, das von einer abstrakten Beschreibung der Anforderungen bis hin zur tatsächlichen Realisierung der Datenbank reicht. Mit diesem Vorgehen möchte man erreichen, daß die Anforderungen, die vom Anwender an eine Datenbank gestellt werden, auch berücksichtigt werden.

Diese Anforderungen zielen vor allem auf einen einfachen, effizienten Zugriff auf alle benötigten Daten ab, d.h. zunächst sollten alle Daten, die von betrieblichen Anwendungen benötigt werden, aus den gespeicherten Daten abgeleitet werden können. Dabei sollte eine redundante Haltung vermieden werden, um Speicherplatz zu sparen und

Anomalien zu umgehen. Solche Anomalien könnten zu Inkonsistenzen in den Daten führen, so daß die hierauf zugreifenden Anwendungen falsche Werte erhalten.

Einen Überblick über die einzelnen Phasen des Datenbankentwurfes und die in jeder Phase erstellten Dokumente gibt die folgende Abbildung. Dabei wäre eine solche Vorgehensweise der Idealzustand. In der Praxis vollzieht sich der Datenbankentwurf häufig weniger strukturiert, wenn anfangs kleinen, überschaubaren Systemen im Laufe der Zeit immer mehr Funktionalität hinzugefügt wird oder eine Ausweitung auf andere betriebliche Bereiche erfolgt. Für das ursprüngliche, überschaubare System wird dann oftmals der Datenbankentwurf auf den Entwurf des internen Schemas beschränkt, so daß über den logischen oder gar den konzeptionellen Entwurf keine Informationen vorhanden sind. So entstehen schlecht strukturierte, unüberschaubare Legacy-Systeme, deren Wartung zunehmend schwieriger, wenn nicht sogar unmöglich wird. Zudem werden Änderungen, die bei bereits im Einsatz befindlichen Systemen notwendig sind, in der Praxis meist nur am internen Schema der Datenbank vorgenommen, eine Anpassung der in den vorangegangenen Phasen erstellten Dokumente bleibt in der Praxis meist aus, so daß hier keine ausreichenden Informationen über die aktuell in den Anwendungen genutzten Datenstrukturen vorliegen. In der Abbildung ist diese Vorgehensweise durch die gestrichelten Pfeile angedeutet.

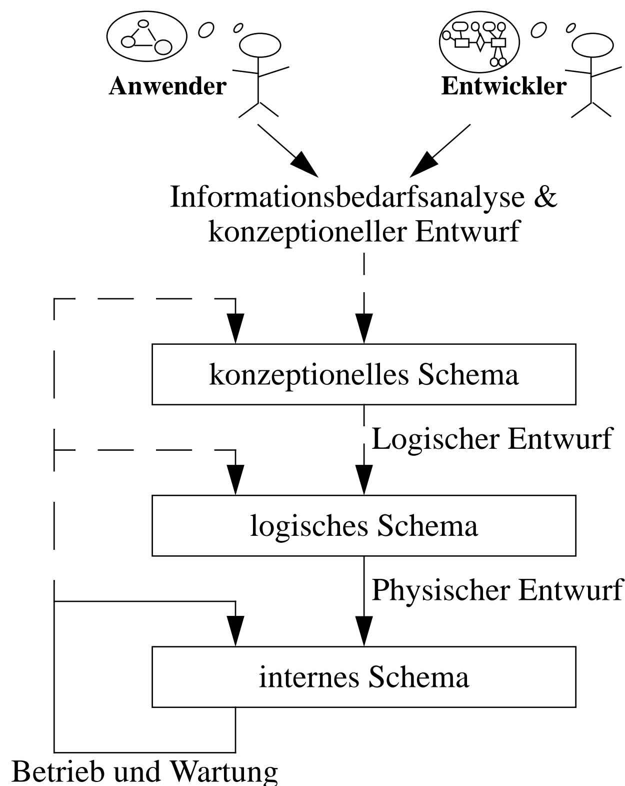


Abbildung 2.1 *Phasenmodell des Datenbankentwurfs*

In der Informationsbedarfsanalyse wird in Zusammenarbeit mit den Anwendern in den Fachabteilungen des Unternehmens eine Analyse über Erwartungen und Anforderungen an das auf die Datenbank aufbauende Informationssystem durchgeführt. Dabei sollte allerdings nicht nur die Frage nach der gewünschten Funktionalität im Vordergrund stehen, sondern auch, welche Werte persistent abgespeichert werden sollen und wie häufig diese abgefragt werden. Ziel dieser Phase ist es, eine informelle Beschreibung der Anforderungen zu erstellen, die dann in den weiteren Phasen der Verständigung zwischen den Beteiligten dienen soll.

In der folgenden Phase wird anhand der gesammelten Anforderungen eine formale Spezifikation der Datenbankinhalte und des Verhaltens vorgenommen. Ziel dieser Phase ist es, ein erstes konzeptionelles Datenbankschema zu erstellen, das unabhängig vom später eingesetzten Datenbank Management System (DBMS) und Datenmodell (z.B. relationales Datenmodell, objektorientiertes Datenmodell) ist. Es wird deshalb ein semantisches Datenmodell entwickelt, das zumeist als ER- (Entity Relationship) oder EER-(Enhanced Entity Relationship) Modell beschrieben wird. ER-Modelle ermöglichen eine abstrakte, darstellungsunabhängige Beschreibung von Datenobjekten und deren Beziehungen. Hierauf wird im Kapitel 2.1.2 anhand des Beispiels noch näher eingegangen.

Wenn sich Anwender und Entwickler auf ein konzeptionelles Schema geeinigt haben, ist dieses auf ein Datenmodell abzubilden. Wie in der Einleitung bereits erwähnt, ist dies in der Praxis meist das Relationenmodell. Hierauf wird im Kapitel 2.1.3 anhand des Beispiels näher eingegangen.

Das so erhaltene logische Datenbankschema wird dann im physischen Entwurf um Angaben zur effizienten Speicherung erweitert. Hierzu zählen Zugriffspfade, Speicherbedarf und auch die Häufigkeit von Anfragen. Das Ergebnis des physischen Entwurfs ist das interne Datenbankschema. Dieses interne Schema wird dann in der Implementierungsphase mit Hilfe der Datenbanksprache (DDL - Data Definition Language) eines konkreten DBMS angelegt. In der Betriebs- und Wartungsphase wird die Datenbank dann mit den von den Anwendungen benötigten Daten gefüllt, diese werden geändert, abgefragt und gegebenenfalls wieder gelöscht. Anpassungen an geänderte Rahmenbedingungen oder Änderungswünsche des Anwenders werden ebenfalls vorgenommen. Hierzu ist es notwendig, die Konzeption der Datenbank neu zu überdenken und die Dokumente der vorangegangenen Phasen nochmals auf ihre Richtigkeit zu prüfen bzw. eine ganz neue Konzeption vorzunehmen.

2.1.1 Beispielszenario

Anhand des folgenden Beispielproblems soll in den weiteren Abschnitten der konzeptionelle und logische Entwurf einer relationalen Datenbank beschrieben werden. Da es sich bei dem Beispiel lediglich um ein relativ überschaubare Problemstellung handelt, kann es allerdings nicht alle Aspekte des Datenbankentwurfs widerspiegeln.

An einer Universität werden von Professoren verschiedene Vorlesungen angeboten. Dabei wird jede Vorlesung von einem Professor gehalten. Jeder Professor hat mehrere Mitarbeiter, die zu den Vorlesungen vertiefende Übungen leiten. Es werden zu jeder Vorlesung Übungen zu verschiedenen Themen angeboten. Jede Übung wird von einem bestimmten Mitarbeiter betreut. Studenten, die die Vorlesung hören, besuchen einige der zugehörigen Übungen. Für die Universität ist die Einrichtung einer Übung allerdings erst möglich, wenn mindestens 10 Studenten daran teilnehmen.

2.1.2 Konzeptioneller Entwurf

Der konzeptionelle Entwurf soll dazu dienen, die informelle Beschreibung der Anforderungen exakt und vollständig zu spezifizieren. Hierzu wird ein semantisches Datenmodell entwickelt, das meist als (E)ER-Modell beschrieben wird. Zu einem gegebenen Problem gibt es allerdings mehrere alternative Modelle. Welches letztendlich Verwendung findet, ist eine subjektive Entscheidung der beteiligten Personen.

Das ER-Modell ist eine grafische Notation, in der die Daten als Entitäten, Attribute und Relationen dargestellt werden.

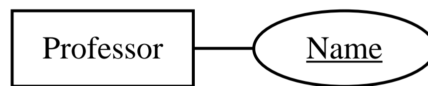
(1) Entitäten (Objekte)

Eine Entität ist ein reales oder gedanklich komplex strukturiertes Objekt aus der zu modellierenden Anwendungsumgebung. Sie sind in der Regel nicht direkt in der Datenbank darstellbar.

Professor

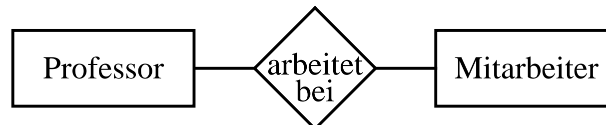
(2) Attribute

Unter Attributen versteht man Eigenschaften der Objekte. Sie stellen unstrukturierbare, atomare Objekte oder Werte dar. Attribute, die ein Objekt eindeutig identifizieren, werden als Schlüssel bezeichnet und durch Unterstreichung gekennzeichnet.



(3) Relationen

Objekte existieren nicht isoliert. Sie stehen in Beziehung zueinander. Diese Beziehungen werden als Relationen bezeichnet.



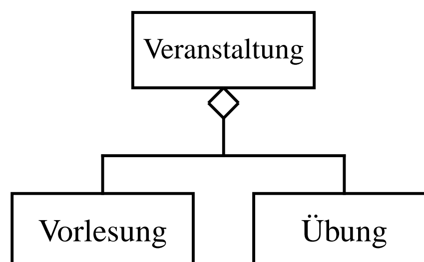
Relationen lassen sich durch folgende Eigenschaften beschreiben:

- *Kardinalität*
  Die Kardinalität einer Relation beschreibt, wieviele Entitäten zu anderen in Beziehung stehen dürfen. Man unterscheidet 1:1-, 1:N-, N:1- und N:M-Beziehungen.
- *Totalität*
  Die Totalität gibt an, ob alle Instanzen eines Objektes in der angegebenen Beziehung stehen müssen, oder ob es Instanzen gibt, die nicht an einer solchen Beziehung beteiligt sind.

Das EER-Modell ist eine Erweiterung des ER-Modells. Hinzu kommen Abstraktionsmechanismen, mit denen hierarchische Strukturen überschaubar modelliert werden können.

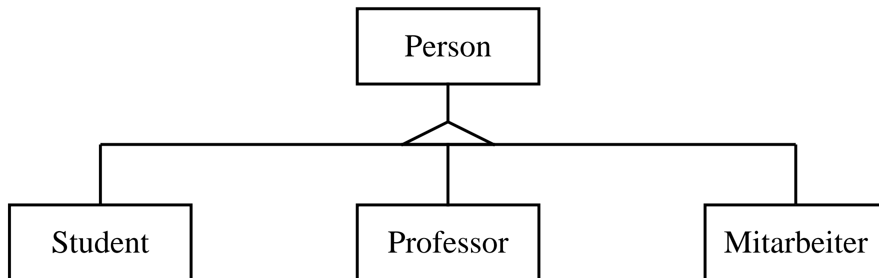
(1) Aggregation

Unter Aggregation versteht man die Zusammenfassung einer bestimmten Zahl von kleineren Einheiten bestimmter Attribute oder Entitäten zu einer neuen Einheit (Entität).



(2) Generalisierung

Die Generalisierung ist eine objektorientierte Modellierungstechnik. Sie erlaubt die Zusammenfassung von gleichen Eigenschaften ähnlicher Objekte zu neuen Objekten. Das neue Objekt wird generalisiert. Die ursprünglichen Objekte sind die Spezialisierung des neuen Objektes. Spezialisierte Entitäten erben alle Attribute der generalisierten Entität.



Die konzeptionelle Modellierung ist ein Prozeß, der kein eindeutiges Ergebnis liefert. Ein konzeptionelles Schema entsteht vielmehr durch ständige Kommunikation zwischen den beteiligten Entwicklern und Anwendern. Für das Beispielszenario habe man sich auf das Schema, das in Abbildung 2.2 zu sehen ist, geeinigt. Zur Vereinfachung wurde auf die Angabe der Attribute der einzelnen Entitäten verzichtet.

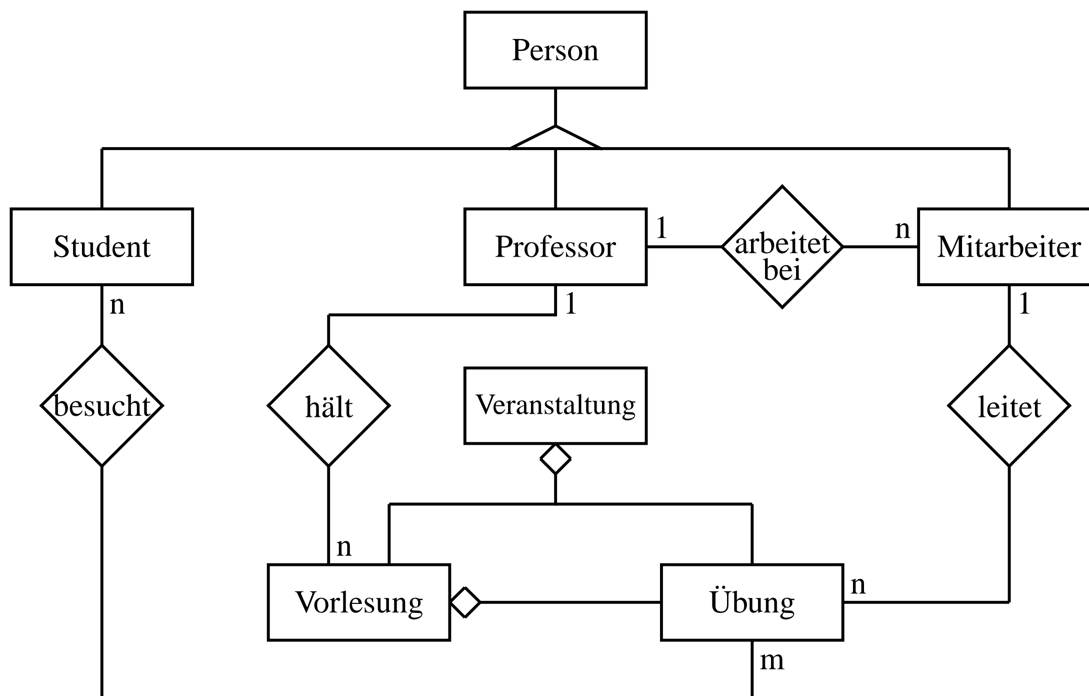


Abbildung 2.2 *Konzeptionelles Schema in ER-Notation (ohne Attribute)*

2.1.3 Relationaler Datenbankentwurf

Ziel des logischen Entwurfs ist es, das vorher entwickelte konzeptionelle Schema auf ein konkretes Datenmodell abzubilden. Für den relationalen Datenbankentwurf bedeutet dies, aus dem konzeptionellen Schema ist ein Relationenschema abzuleiten.

Das Relationenmodell ist Grundlage aller relationaler DBMS. Im Vergleich zum ER-Modell gibt es im Relationenmodell allerdings nur Objekte und Attribute. Beziehungen können durch Verknüpfungen von Relationen modelliert werden.

(1) Werte

Werte entsprechen den Attributen im ER-Modell. Hierbei handelt es sich also um atomare Werte, die durch Standarddatentypen beschrieben werden. Sie sind die kleinste logische Einheit des Relationenmodells.

(2) Relationen / Tupel

Eine Relation wird deklariert durch einen Relationennamen und einer Liste von Attributen als $R(A_1:D_1,...,A_n:D_n)$. Wobei R den Namen der Relation bezeichnet und A_i den Namen des Wertebereiches D_i. Eine Relation entspricht also einem Objekttypen im ER-Modell. Die Menge aller Relationendeklarationen heißt Relationenschema. Eine Relation ist dann eine Menge von n-Tupeln $r=\{t_1,...,t_n\}$. Jedes n-Tupel ist eine Liste aus n Werten mit $t=<v_1,...,v_n>$, wobei v_i aus dem Wertebereich von A_i oder Null sein muß.

Üblicherweise werden Relationen in Tabellenform dargestellt:

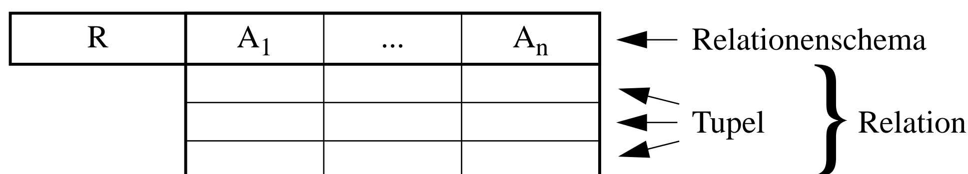


Abbildung 2.3 *Repräsentation von Relationen durch Tabellen*

Eine relationale Datenbank besteht aus mehreren Relationen, deren Tupel miteinander in Beziehung stehen können. Die Reihenfolge der Tupel innerhalb der Relation ist ohne Bedeutung, es darf allerdings kein Tupel mehrfach vorkommen. Um dies zu verhindern sollte für jede Relation ein Primärschlüssel angegeben werden, der eine eindeutige Identifizierung der Tupel ermöglicht. Die Angabe eines Primärschlüssels ist allerdings bei einigen (vor allem älteren) DBMS nicht zwingend vorgeschrieben.

Generell werden Bedingungen, die die Relationenschemata eines DBMS erfüllen sollen, als Integritätsbedingungen bezeichnet. Die Einhaltung der Integritätsbedingungen ist vom DBMS sicherzustellen. Eine dieser Eigenschaften kann z.B. auch die referentielle Integrität der Relationenschemata sein. Diese besagt, daß der Primärschlüssel X einer Relation R_1, wenn er in einem Tupel der Relation R_2 als Attribut[1] referenziert wird, auch in der Ausgangsrelation R_1 als Tupel vorhanden sein muß. Falls der Fremdschlüssel aus mehreren Attributen besteht, müssen natürlich in R_2 auch alle Attribute des Schlüssels aus R_1 auftreten. Das DBMS sorgt dann bei allen Änderungen an den Relationen R_1 und R_2 dafür, daß alle Tupel, die durch eine Fremdschlüsselbeziehung referenziert werden, entsprechend geändert werden bzw. verbietet diese Änderungen. Diese Funktionalität bieten wiederum nicht alle (vor allem ältere) DBMS an.

Zur Übersetzung eines konzeptionellen ER-Schemas in das Relationenmodell wird jede Entität $E(\underline{X_1,..,X_m},A_1,..,A_n)$ mit den Schlüsselattributen $X_1,..,X_m$ in ein Relationenschema $R(\underline{X_1,..,X_m},A_1,..,A_n)$ überführt. Für das Beispiel bedeutet dies, wir erhalten die Relationenschemata `Person`, `Professor`, `Mitarbeiter`, `Student`, `Veranstaltung`, `Vorlesung` und `Übung` mit den entsprechenden Schlüssel- und Nichtschlüsselattributen. Häufig werden der Einfachheit halber künstliche Schlüsselattribute eingeführt, die nicht notwendig wären. Einige DBMS verlangen die Definition von eindeutigen Indizes, um so Primärschlüssel nachzubilden. Allerdings können diese Indizes Nullwerte enthalten, was der Definition von Primärschlüsseln widerspricht [PB95].

Die hierarchischen Strukturen, die im EER-Modell durch Aggregation und Generalisierung entstehen, können im Relationenmodell nicht abgebildet werden, so daß diese Strukturen nachgebildet werden müssen. Konkret heißt dies, daß Entitäten aufgespalten werden in separate Super- und Subklassentabellen oder mehrere Entitäten werden in mehreren Sub- bzw. einer Superklassentabelle zusammengefaßt, so daß die benutzten Modellierungskonzepte verlorengehen. Alle möglichen Realisierungsalternativen bringen jedoch Nachteile mit sich. So widerspricht die Zusammenfassung der Generalisierung in mehreren Subklassentabellen sowie die Aufspaltung in separate Super- und Subklassentabellen der Philosophie der Normalisierung, die besagt, daß zusammengehörende Informationen an einer einzigen Stelle gespeichert werden (vgl. [PB95]). Die Variante, die Attribute der Superklasse und aller Subklassen in einer Superklassentabelle zusammenzufassen, widerspricht der 2. Normalform (vgl. Beschreibung der Normalisierung später in diesem Kapitel).

1. X ist in Relation R_2 ein Fremdschlüssel

Im Beispiel entscheidet man sich für die Variante, mehrere Subklassentabellen anzulegen. Die Generalisierung von `Student`, `Professor` und `Mitarbeiter` zu `Person` führt somit dazu, daß die spezialisierten Schemata `Student`, `Professor` und `Mitarbeiter` alle Attribute der generalisierten Entität `Person` übernehmen und zudem noch einige spezielle Attribute enthalten. Ein Schema für die Entität `Person` wird also nicht angelegt. Da Studenten eigentlich keine Personalnummer (Pers.-Nr.) besitzen, entscheidet sich der Datenbankentwickler außerdem, den Schlüssel der Relation `Student` in Matr.-Nr. umzubenennen.

Bei der Aggregation von `Vorlesung` und `Übung` zu einer `Veranstaltung` vollzieht sich die Überführung in das Relationenmodell ähnlich. Hier enthält das Relationenschema `Veranstaltung` neben einer Ordnungsnummer als Schlüsselattribut lediglich die Fremdschlüssel der aggregierten Entitäten, um so die Zuordnung von Vorlesungen zu Übungen möglich zu machen.

Aus den Beziehungstypen eines ER-Modells werden Relationenschemata abgeleitet, die die Schlüsselattribute aller an der Beziehung beteiligten Entitäten sowie alle weiteren Attribute der Beziehung enthalten. Somit werden im Beispiel noch folgende Relationenschemata angelegt: `arbeitet_bei`, `hält`, `besucht` und `leitet`. Bei 1:N-Beziehungen ist es üblich, keine extra Relation anzulegen, sondern den Schlüssel der 1-seitigen Relation als Fremdschlüssel in die N-seitigen Relation mit aufzunehmen, so daß diese Beziehung nicht ohne weiteres aus den Relationenschemata zu erkennen ist. Im Beispiel kann so die Definition eigener Relationenschemata für die Beziehungen `arbeitet_bei`, `hält` und `leitet` umgangen werden. 1:1-Beziehungen können auf die gleiche Weise abgebildet werden. Es kommt laut [PB95] auch gelegentlich vor, daß 1:1-Beziehungen im Relationenmodell realisiert werden, indem Fremdschlüssel in beiden Relationen definiert werden, so daß von beiden Seiten ein schneller Zugriff möglich ist. Für das Reverse Engineering ist die konkrete Realisierungsvariante zunächst unbekannt und muß durch Analysen mühsam herausgefunden werden.

Ein mögliches Ergebnis des oben für das Beispiel beschriebenen Entwurfsschrittes können die folgende Relationenschemata sein:

Professor:

Pers-Nr	Name	Adresse	Fachbereich	FName	Lehrstuhl

Mitarbeiter:

Pers-Nr	Name	Adresse	Fachbereich	FName	Raumnr.	Professor

Student:

Matr-Nr	Name	Adresse	Fachbereich	FName	Stud.gang	Semester

Veranstaltung:

V-Nr	Vorl-Nr	Übg-Nr

Vorlesung:

Vorl-Nr	Bez.	Professor

Übung:

Übg-Nr	Thema	Leiter

besucht:

Matr-Nr	Übg-Nr

Abbildung 2.4 *Relationenschemata für das Beispielszenario*

Durch die Abbildung des konzeptionellen Schemas auf das Relationenmodell gehen also einige semantische Informationen verloren bzw. sie sind nicht mehr explizit modelliert, da sie vom Relationenmodell nicht unterstützt werden wie z.B. Generalisierung oder Aggregation. Wie anhand des Beispiels beschrieben, gibt es außerdem mehrere Möglichkeiten ein konzeptionelles Schema in das Relationenmodell umzusetzten. Das Reverse Engineering muß versuchen diese Informationen aus einem vorliegenden Relationenmodell wiederzugewinnen.

Die aus dem EER-Modell abgeleiteten Relationenschemata enthalten i.d.R. allerdings einige nicht wünschenswerte Eigenschaften. Falls sich z.B. der Name eines Fachbereiches (`FName`) ändert, muß diese Änderung in allen entsprechenden Tupeln der Schemata `Student`, `Professor` und `Mitarbeiter` erfolgen. Hier gibt es also noch redundante Informationen. Die *Normalisierung* der Schemata soll deshalb dazu beitragen, solche Schwachstellen ausfindig zu machen und die Schemata durch „bessere“, äquivalente Schemata darzustellen.

In der betrieblichen Praxis vollzieht sich die Normalisierung i.d.R. in 3 Schritten. Die vorhandenen Schemata werden auf 3 sogenannte Normalformen (NF) hin überprüft und diese gegebenenfalls durch Umstrukturierungen in den Schemata herbeigeführt. Bei der Normalisierung geht man davon aus, daß zunächst alle Schemata in der 1. NF vorliegen, d.h. die Schemata enthalten nur atomare Werte. Dies ist auch im Beispiel der Fall.

In der 2. und 3. NF wird durch das Auffinden von sog. *funktionalen Abhängigkeiten* versucht, Redundanzen zu vermeiden. „Eine funktionale Abhängigkeit gilt dann innerhalb einer Relation zwischen Attributmengen X und Y, wenn in jedem Tupel der Relation der Attributwert unter den X-Komponenten den Attributwert unter den Y-Komponenten festlegt. Unterscheiden sich also zwei Tupel in den X-Attributen nicht, so haben sie auch gleiche Werte für alle Y-Attribute. Die funktionale Abhängigkeit [...] wird dann mit $X \rightarrow Y$ bezeichnet“ [HS95]. Ein Schema ist in der 2. NF, wenn alle Attribute voll funktional abhängig vom gesamten Schlüssel des Schemas sind. Im Beispiel ist diese Bedingung für alle Schemata erfüllt.

In der 3. NF wird, wenn das Schema in der 2. NF ist, überprüft, ob Attribute eines Schemas nur transitiv vom Schlüssel des Schemas abhängt. In den Beispielschemata `Student`, `Professor` und `Mitarbeiter` z.B. hängt der Name des Fachbereiches (`FName`) vom Attribut `Fachbereich` ab, welches vom Schlüssel `Pers-Nr` (bzw. `Matr-Nr`) nicht trivial abhängt. Es existiert hier also die transitive Abhängigkeit $\texttt{Pers-Nr} \rightarrow \texttt{Fachbereich} \rightarrow \texttt{FName}$. Um die Schemata nun in die 3. NF zu überführen, werden sie folgendermaßen aufgespalten:

Professor, Mitarbeiter:

<u>Pers-Nr</u>	Name	Adresse	Fachbereich	...

Fachbereich:

<u>Fachbereich</u>	FName

Abbildung 2.5 *Schemata in 3NF*

Das durch die Normalisierung erhaltene Schema sollte frei von Redundanzen und optimal auf die modellierten Anforderungen zugeschnitten sein. In der Literatur werden häufig noch weitere Normalformen (4. NF, 5. NF bzw. BCNF) diskutiert, allerdings wird in der Praxis eine Normalisierung meist nur bis zur 3. NF durchgeführt. Aus diesem Grund sollen weitere Normalformen auch hier nicht betrachtet werden.

2.1.4 Optimierungen beim physischen Datenbankentwurf

Es kommt häufig vor, daß an Relationenschemata, die der 3. NF entsprechen, noch Änderungen vorgenommen werden, bevor sie implementiert werden. Diese Änderungen sollen dazu dienen, typische Anfragen an die Relationen möglichst schnell durchführen zu können.

Solche Optimierungen hängen immer vom jeweiligen Schema und den Anforderungen der Benutzer ab. Es gibt keine allgemeingültige Vorgehensweise, um das Anfrageverhalten einer Datenbank zu verbessern.

Im Beispiel könnte eine typische Anfrage z.B. die Frage nach allen Teilnehmern bestimmter Übungen sein. Eine Antwort hierauf findet man in der Relation `besucht`, in der die Beziehung zwischen `Student` und `Übung` abgebildet wird. Da an einer Übung aber mindestens 10 Studenten teilnehmen, ließen sich diese Anfragen optimieren, indem in der Relation `besucht` neben der Nummer der Übung (`Übg-Nr`) nicht nur ein Student sondern gleich 10 Studenten im Schema berücksichtigt werden. Falls mehr als 10 Studenten an einer Übung teilnehmen (und zur eindeutigen Identifizierung der Tupel), wird zusätzlich das Attribut `Seq` eingeführt, das zunächst für das erste Tupel einer Übung auf eins gesetzt und für weitere Tupel, die die gleiche Übung referenzieren, inkrementiert wird.

Besuch:

Übg-Nr	Seq.	Matr-Nr1	Matr-Nr2	....	Matr-Nr10

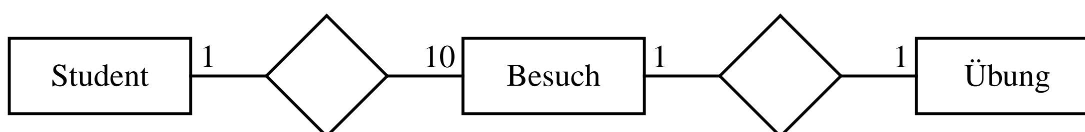


Abbildung 2.6 *Mögliche Optimierungsstruktur einer N:M-Beziehung*

2.2 Die relationale Datenbanksprache SQL

Mit der *Structured Query Language* (SQL) existiert eine von ANSI (American National Standards Institute) und ISO (International Standardization Organization) genormte relationale Datenbanksprache, die von allen Datenbankherstellern unterstützt wird. Sie unterteilt sich in die *Data Definition Language* (DDL) und die *Data Manipulation Language* (DML).

Mit Hilfe der DDL erfolgt die Implementierung der zuvor modellierten internen Datenbankschemata. Die DDL erlaubt das Anlegen, Ändern und Löschen der Relationenschemata in der relationalen Datenbank.

Diese Metadaten der Relationen werden dann im betrieblichen Alltag von den Anwendungen mit Anwenderdaten gefüllt, es werden Änderungen durchgeführt, Daten gelöscht oder Anfragen an die Datenbank gestellt. All diese Operationen werden von der DML zur Verfügung gestellt. Da die Änderungsoperationen häufig eine recht einfache Struktur haben und für die Cliché-Erkennung eine untergeordnete Rolle spielen, soll im folgenden auf die Anfrageoperationen von SQL eingegangen werden.

Eine Anfrage an eine relationale Datenbank liefert als Ergebnis ein gesuchtes Schema, welches durch das `SELECT`-Statement genau spezifiziert wird. Die Syntax einer `SELECT`-Anfrage sieht folgendermaßen aus (vgl. Anhang A, [DAT89]):

`SELECT [ALL|DISTINCT]` *selection*

`FROM` *table_ref_commalist*

`[WHERE` *search_condition*`]`

`[GROUP BY` *column_ref_commalist*`]`

`[HAVING` *search_condition*`]`

Dabei bedeuten:

selection

Die *selection* beschreibt, welche Attribute das Ergebnisschema enthält. Diese Attribute sind eine Teilmenge aller Attribute der Tabellen, die in der `FROM`-Klausel aufgeführt sind. Es können auch arithmetische Operationen oder Aggregatfunktionen angegeben werden.

table_ref_commalist

Die *table_ref_commalist* bezeichnet die Relationen, die zur Auswertung der Anfrage benutzt werden. Von allen angegebenen Relationen wird das kartesische Produkt gebildet.

search_condition

Die *search_condition* dient der Einschränkung der Tupel. Die Ergebnisrelation besteht nur aus Tupeln, die die Bedingungen erfüllen, die in der Qualifikation aufgestellt werden. Dabei bezieht sich die *search_condition* der `HAVING`-Klausel auf die Gruppenbildung.

column_ref_commalist

Die *column_ref_commalist* listet die Attribute auf, nach denen die Ergebnisrelation gruppiert wird.

Auf die Semantik der einzelnen Teile einer `SELECT`-Anfrage soll hier nicht weiter eingegangen werden. Ausführliche Informationen hierzu befinden sich z.B. in [DAT89], [EN94] oder [HS95]. Einige, gerade für das Reverse Engineering interessante Anfragen werden im Kapitel 3 genauer betrachtet.

Da mit SQL eine genormte relationale Datenbanksprache existiert, die von allen namhaften relationalen Datenbanken unterstützt wird, befaßt sich eine Cliché-Erkennung in relationalen Datenbankanwendungen natürlich primär mit der Auswertung der in den Anwendungen genutzten SQL-Anfragen. Diese Anfragen werden aus Anwendungen aufgerufen, die in einer anderen Programmiersprache geschrieben sind. Diese sog. *Embedded SQL*-Anweisungen werden mit Hilfe eines Vorübersetzers (engl. Precompiler) in Prozeduraufrufe der entsprechenden Datenbank umgewandelt. Dabei sucht der Vorübersetzer nach den notwendigen Schlüsselworten `EXEC SQL`, die vor den eigentlichen Datenbankaufrufen stehen müssen. Dieses Einbetten von Datenbankzugriffen in andere Programmiersprachen wurde erst 1992 in die SQL-Normung aufgenommen, obwohl diese Möglichkeit schon früher angeboten wurde.

Eine erste SQL-Norm stammt aus dem Jahr 1986, sie wurde 1989 um sogenannte *Integrity Enhancement Features* (IEF) erweitert (SQL-89), bevor 1992 dann einer neuer SQL-Standard als SQL2 (SQL-92) veröffentlicht wurde. In vielen Datenbankanwendungen wird derzeit meist der SQL1-Standard (SQL-89) unterstützt. Der aktuelle, um einige Funktionalitäten erweiterte Standard ist aber SQL2 (oder SQL-92). Dies führt beim Reverse Engineering zu dem Problem, daß in betrieblichen Anwendungen wahrscheinlich beide Standards zu finden sind. Es kann nicht davon ausgegangen werden, daß ältere Anwendungen so umgestellt wurden, daß sie die Funktionalität der neuen Standards voll ausnutzen. Auf einige für das Reverse Engineering relevante Unterschiede zwischen dem SQL1- und SQL2-Standard soll nun eingegangen werden.

Die Erweiterungen, die SQL-89 gegenüber dem Standard von 1986 hinzugefügt wurden, umfassen Komponenten zur Integritätssicherung (IEF) [HS95]. Im einzelnen sind dies, die Möglichkeit Primär- und Fremdschlüssel anzugeben (durch die Schlüsselworte `PRIMARY KEY (Attribut)` bzw. `FOREIGN KEY (Attribut) REFERENCES Tabelle(Attribut)`) und die Angabe von Integritätsbedingungen in `CHECK`-Klauseln bei der Definition *einer* Tabelle.

Mit SQL-92 ist es nun möglich diese Integritätsbedingungen für *mehrere* Tabellen zu definieren. Desweiteren kann der Verbund der Tabellen in der `FROM`-Klausel auf verschiedene Arten durchgeführt werden. In SQL-89 wurde immer das kartesische Produkt gebildet, in SQL-92 können verschiedene Varianten gewählt werden, wie z.B. `CROSS JOIN`, `NATURAL JOIN`, `OUTER JOIN` u.a. In älteren SQL-Versionen war es außerdem nicht möglich, beliebige `SELECT`-Klauseln in die `FROM`-Klausel einer Anfrage einzubetten. Dies ist in SQL-92 möglich.

3 Reverse Engineering

Das Reverse Engineering befaßt sich mit der Analyse und Redokumentation vorhandener Legacy-Systeme, das sind Informationssysteme, die teilweise bereits jahrzehntelang im Einsatz sind. Durch anfallende Änderungen wurde das ursprüngliche Design dieser Systeme immer wieder geändert. Aus den unterschiedlichsten Gründen ist aber keine Anpassung der Dokumentation erfolgt bzw. keine Dokumentation erstellt worden. In diesem Kapitel wird ein Überblick über das Reverse Engineering von relationalen Datenbanken gegeben. Dabei werden zunächst einmal die möglichen Ansatzpunkte vorgestellt. Anhand der in Kapitel 2 entwickelten Relationenschemata sollen dann diese Ansatzpunkte beispielhaft betrachtet werden. Abschließend wird auf generelle Probleme des Reverse Engineerings von relationalen Datenbanken eingegangen.

3.1 Ansatzpunkte für das Reverse Engineering

In Kapitel 2 wurde gezeigt, wie schon im Verlaufe des Datenbankentwurfsprozesses semantische Informationen verlorengehen. Durch jahrelange Nutzung einer Datenbank und den daraus resultierenden Änderungen und Anpassungen, die von den unterschiedlichsten Entwicklern mit den verschiedensten Kenntnissen und Fähigkeiten durchgeführt wurden, ist eine Wiedergewinnung dieser verlorengegangenen semantischen Informationen natürlich nicht einfacher geworden.

In datenorientierten Anwendungen kann der Prozeß des Reverse Engineering allerdings leicht aufgeteilt werden, so daß er an Komplexität verliert. Hier kann die Struktur und Semantik der Datenbank meist unabhängig von den zugreifenden Anwendungen analysiert werden. Nach [HEH+98] bezeichnet man eine solche Vorgehensweise als *Datenbank Reverse Engineering* (engl. database reverse engineering, DBRE). Dabei ist es empfehlenswert zunächst die benutzten Datenstrukturen zu betrachten, denn hier gestaltet sich die Suche nach der zugrundeliegenden Semantik meist einfacher, als in den teilweise komplexen Algorithmen der Anwendungen. Zudem sind die persistenten Datenstrukturen oftmals der stabilste Teil der Anwendungen, d.h. die Änderungshäufigkeit der Strukturen ist wesentlich geringer als die der Anwendungen. Dies mag eine Folge des großen Änderungsaufwandes sein, den eine Änderung der Datenstrukturen an allen zugreifenden Anwendungen nach sich zieht. Ein weiterer Grund, die Analyse der

persistenten Datenstrukturen an den Anfang des Reverse Engineering-Prozesses zu stellen, ist die Tatsache, daß es einfacher ist, die Abläufe und Abhängigkeiten innerhalb einer Anwendung zu ergründen, wenn man die benutzten Datenstrukturen und deren Bedeutung kennt.

Demzufolge sollte der erste Ansatzpunkt für das Reverse Engineering das interne Schema, also die Metadaten, der Datenbank sein. Weitere Informationen über die vorhandenen Strukturen in der Datenbank erlangt man durch die Betrachtung der vorhandenen Anwenderdaten in der Datenbank und durch die Analyse der Datenbankzugriffe in den Anwendungen. Hieraus läßt sich dann ein sogenanntes *semantisch angereichertes Datenbankschema* erstellen, das einen detaillierten Einblick in die Semantik der Datenstrukturen gibt. Im nächsten Reverse Engineering-Schritt kann dann mit Hilfe dieser Informationen eine Analyse der Semantik der zugreifenden Anwendungen erfolgen.

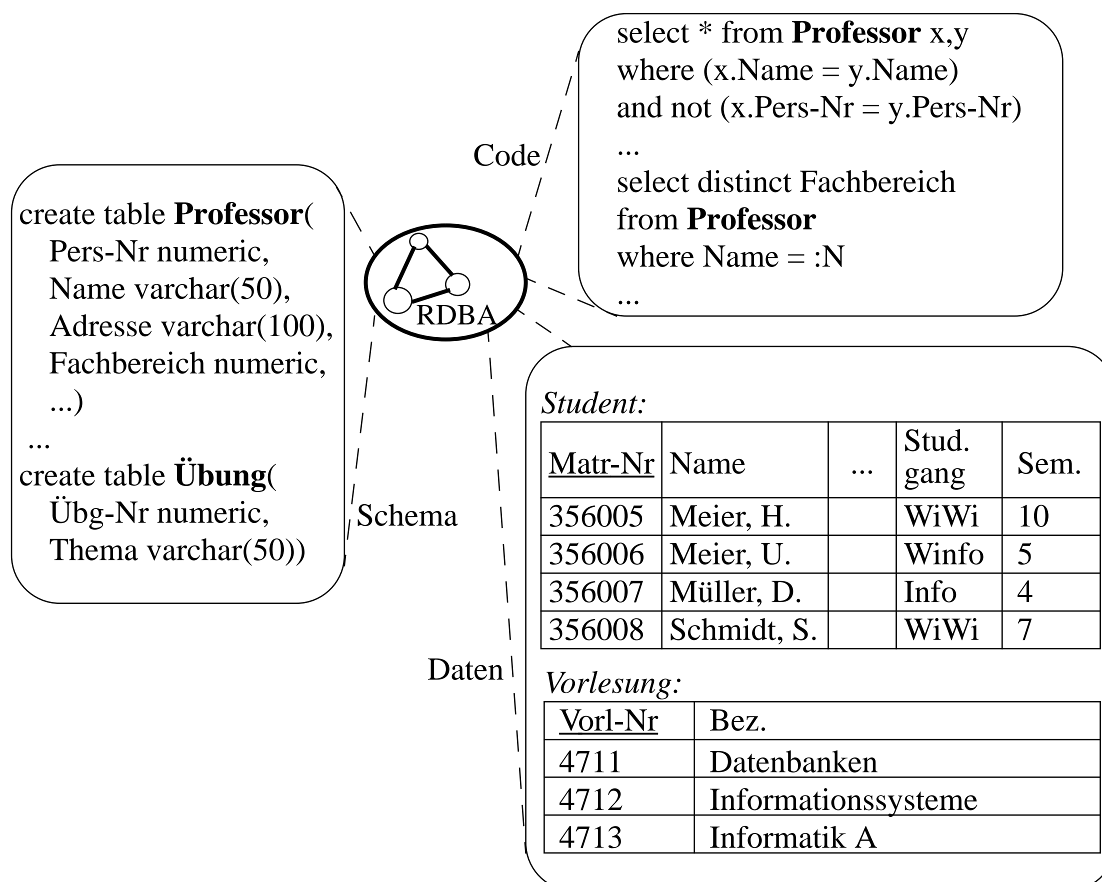


Abbildung 3.1 *Ansatzpunkte für das Datenbank Reverse Engineering*

Die Vorstellung des DBRE soll sich hier aber auf die Analyse der Datenstrukturen beschränken. Einen weiteren Überblick über die Vorgehensweise beim DBRE geben [FV95], [JSZ97] oder [HEH+98]. Die Reihenfolge der nächsten Kapitel soll nicht den Ablauf des DBRE-Prozesses widerspiegeln. Hiermit soll vielmehr ein Eindruck vermittelt werden, welche Informationen aus den einzelnen Teilen einer relationalen Datenbankanwendung zu gewinnen sind.

3.2 1. Ansatzpunkt: Datenbankschema

Einen ersten Ansatzpunkt für das Reverse Engineering bildet das implementierte Datenbankschema. Das in Kapitel 2 erstellte Datenbankschema soll hier als Grundlage für eine Analyse dienen. Es besteht aus den folgenden Relationenschemata mit den gegebenen Attributen und Datentypen.

Professor	*(Pers-Nr:numeric, Name:varchar(50), Adresse:varchar(100), Fachbereich:numeric, Lehrstuhl:varchar(50))*
Mitarbeiter	*(Pers-Nr:numeric, Name:varchar(50), Adresse:varchar(100), Fachbereich:numeric, Raumnr:numeric, Professor:numeric)*
Student	*(Matr-Nr:numeric, Name:varchar(50), Adresse:varchar(100), Fachbereich:numeric, Studgang:varchar(20), Semester:numeric)*
Veranstaltung	*(Vorl-Nr:numeric, Übg-Nr:numeric)*
Vorlesung	*(Vorl-Nr:numeric, Bez:varchar(50), Professor:numeric)*
Übung	*(Übg-Nr:numeric, Thema:varchar(50), Leiter:numeric)*
Besuch	*(Übg-Nr:numeric, Seq:numeric, Matr-Nr1:numeric,..,Matr-Nr10:numeric)*
Fachbereich	*(Fachbereich:numeric, FName:varchar(50))*

Abbildung 3.2 *Beispielschema mit Attributen und zugehörigen Datentypen*

Hieraus lassen sich zunächst einmal Informationen über die Strukturen der Daten innerhalb einer Tabelle gewinnen. So kann der DDL-Code z.B. explizit Schlüssel- und Fremdschlüsselangaben oder Indexdefinitionen enthalten, so daß Schlüsselkandidaten einfach zu erkennen sind. In früheren SQL-Versionen waren diese Angaben aber teilweise noch nicht möglich oder werden auch heute nicht unbedingt von allen Entwicklern genutzt, so daß hier eine weitere Analyse notwendig ist. Attribute, die im Schema mit dem Zusatz `NOT NULL` gekennzeichnet sind, geben ebenfalls einen

Hinweis darauf, daß sie Schlüsselkandidaten sind. An den gegebenen Tabellendefinitionen des Beispiels läßt sich jedoch über Schlüsselkandidaten zunächst nichts ablesen.

Auf jeden Fall ist jedem Attribut ein Name und ein Datentyp zugeordnet. Attribute mit ähnlichen Namen und Datentypen in verschiedenen Tabellen lassen dann darauf schließen, daß sie die gleiche Bedeutung haben, so daß hier eventuell eine Fremdschlüsselbeziehung vorliegt. Die Attribute `Pers-Nr` im Schema `Professor` und das Attribut `Professor` der Tabelle `Mitarbeiter` liefern ein Beispiel hierfür.

Die Tabellen `Mitarbeiter`, `Professor` und `Student` besitzen die Attribute `Name`, `Adresse` und `Fachbereich` mit gleichen Namen und gleichen Datentypen. Auch dies deutet zunächst auf eine Fremdschlüsselbeziehung hin. Analog gilt dies noch für weitere Attribute, die unten aufgelistet sind.

Als Ergebnis der Schemaanalyse erhalten wir also eine Reihe von Attributen der verschiedenen Tabellen mit gleichen oder ähnlichen Namen und gleichen oder ähnlichen Datentypen, die man in die folgenden sogenannte *Äquivalenzklassen* [FV95] unterteilen kann.

{Professor.Pers-Nr, Mitarbeiter.Professor, Vorlesung.Professor}

{Professor.Pers-Nr, Mitarbeiter.Pers-Nr}

{Professor.Name, Mitarbeiter.Name, Student.Name}

{Professor.Adresse, Mitarbeiter.Adresse, Student.Adresse}

{Professor.Fachbereich, Mitarbeiter.Fachbereich, Student.Fachbereich, Fachbereich.Fachbereich}

{Student.Matr-Nr, Besuch.Matr-Nr1,.., Besuch.Matr-Nr10}

{Veranstaltung.Vorl-Nr, Vorlesung.Vorl-Nr}

{Veranstaltung.Übg-Nr, Übung.Übg-Nr, Besuch.Übg-Nr}

***Abbildung 3.3** Äquivalenzklassen des Beispielschemas*

3.3 2. Ansatzpunkt: Datenbankausprägungen

Anhand der in den Tabellen vorhandenen Anwenderdaten können die oben erhaltenen Äquivalenzklassen genauer untersucht werden, um die Hinweise auf Fremdschlüsselbeziehungen weiter zu verstärken oder zu entkräften. Ob es sich bei bestimmten Attributen

tatsächlich um Schlüsselkandidaten handeln kann, läßt sich z.B. schnell widerlegen, wenn in den Daten eines Attributes gleiche Einträge gefunden werden.

Anhand der folgenden Beispielausprägungen einiger Tabellen des Beispielschemas werden nun weitere Analysemöglichkeiten beschrieben:

Professor:

Pers-Nr	Name	Adresse	Fachbereich	...
123400	Meier, Hans	Bahnhofstr. 45, 33098 PB	17	
123401	Stahlmann, Peter	Ahornallee 3, 33104 PB	2	
123402	Zacharias, Eva	Berliner Str. 120, 33100 PB	7	

Mitarbeiter:

Pers-Nr	Name	Adresse	Fachbereich	...
257801	Mahlzahn, Martin	Lummerweg 2, 33102 PB	5	
257802	Knopf, Jim	Emmastr 33, 33334 GT	12	
257803	Zacharias, Eva	Paderborner Str. 67, 33200 BI	3	

Student:

Matr-Nr	Name	Adresse	Fachbereich	...
356001	Bieler, Brigitte	Ahornallee 3, 33104 PB	5	
356002	Meier, Hans	Pappelweg 9, 33310 GT	7	
356003	Westfalen, Vera	Niedernwall 25, 33200 BI	17	

Abbildung 3.4 *Beispielausprägungen in den Tabellen Professor, Mitarbeiter und Student*

Beim Blick auf die Ausprägungen der Tabellen `Mitarbeiter`, `Professor` und `Student` wird man schnell zu dem Schluß kommen, daß die im vorherigen Kapitel erkannten Äquivalenzklassen keine Fremdschlüsselbeziehung darstellen, da die Attribute `Name`, `Adresse` und `Fachbereich` disjunkt sind. Es existieren zwar einige gleiche Ausprägungen in den jeweiligen Spalten der Tabellen, aber um eine Fremdschlüsselbeziehung nachzuweisen, sollten alle Einträge aus zwei Tabellen in der dritten vorhanden sein. Gleiches gilt für die Äquivalenzklasse `Professor.Pers-Nr` und `Mitarbeiter.Pers-Nr`. Hieraus läßt sich eher schließen, daß in den drei Tabellen eine Vererbungsbeziehung versteckt ist. Da in der Tabelle `Student` das Attribut `Matr-Nr` den gleichen Datentyp wie die `Pers-Nr` in den Tabellen `Professor` und `Mitarbeiter` hat und es sich zudem offensichtlich in allen Fällen um eine (Ordnungs-) Nummer handelt, kann man auch dieses Attribut in die Vererbung mit einbeziehen.

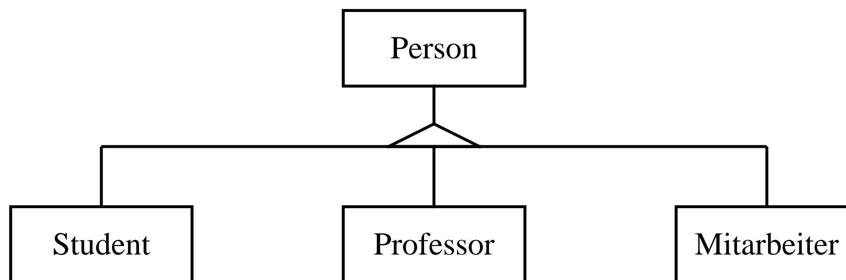


Abbildung 3.5 *Anhand der Datenbankausprägungen abgeleitete Vererbungsbeziehung*

3.4 3. Ansatzpunkt: Datenbankzugriffe

Eine weitere Analyse läßt sich dann mit Hilfe der in den Anwendungen genutzten SQL-Anfragen durchführen. Das Beispielschema soll hierzu anhand einiger Beispielanfragen analysiert werden.

```
select distinct Fachbereich
  from Professor
  where Name = :N
...
select *
   from  Professor x,y
   where (x.Name = y.Name)
     and not (x.Pers-Nr = y.Pers-Nr)
...
select s.Name
  from Student s, Besuch b, Übung u
  where u.Thema = :T
    and u.Übg-Nr = b.Übg-Nr
    and ( b.Matr-Nr1 = s.Matr-Nr or
          b.Matr-Nr2 = s.Matr-Nr or
          ...
          b.Matr-Nr10 = s.Matr-Nr)
```

Abbildung 3.6 *Beispielanfragen*

Die erste Anfrage sucht nach dem Fachbereich eines bestimmten Professors. Sie enthält das SQL-Schlüsselwort `distinct`, d.h. es wird erwartet, daß mehrere gleiche Tupel gefunden werden, die aber nicht doppelt zurück gegeben werden sollen. Hieraus kann

man demzufolge ableiten, daß die Attribute `Fachbereich` und `Name` der Tabelle `Professor` gleiche Werte enthalten können und somit als Schlüsselkandidaten ausscheiden. Es kann natürlich vorkommen, daß ein Entwickler alle Anfragen mit der `distinct`-Angabe versieht, so daß die Informationen, die durch diese Analyse gewonnen werden, nicht richtig sind und man in weiteren Analysen zu gegensätzlichen Ergebnissen kommt. Hier ist das Eingreifen eines Experten wichtig, der die bisher gewonnen Ergebnisse überprüft, bevor weitere Analysen gestartet werden.

Mit der zweiten Anfrage werden alle Professoren gesucht, deren Namen in der Tabelle mehrfach vorkommen. Dazu wird ein Verbund über die gleiche Tabelle durchgeführt. Mit der Bedingung `not (x.Pers-Nr = y.Pers-Nr)` soll sichergestellt werden, daß der Verbund nicht über gleiche Tupel erfolgt, sondern nur über Tupel, bei denen das Attribut `Name` gleich ist. Die eindeutige Identifizierung der Tupel erfolgt also über das Attribut `Pers-Nr`, und somit ist dieses Attribut ein Schlüsselkandidat.

Das Ergebnis der dritten Anfrage sind die Namen aller Studenten, die an einer Übung zu einem bestimmten Thema teilgenommen haben. Dabei weist die Bedingung `u.Übg-Nr = b.Übg-Nr` zunächst einmal darauf hin, daß eine Fremdschlüsselbeziehung zwischen `Übung.Übg-Nr` und `Besuch.Übg-Nr` vorliegt. Da diese Bedingung zur eindeutigen Identifizierung der Übung benutzt wird, kann man außerdem schließen, daß das Attribut `Übg-Nr` der Primärschlüssel der Tabelle `Übung` ist. Die zusätzliche Bedingung verstärkt ebenfalls den Hinweis auf eine Fremdschlüsselbeziehung zwischen den Attributen `Besuch.Matr-Nr1,..,Besuch.Matr-Nr10` und `Student.Matr-Nr`. Dabei geht der Vergleich jeweils über das gleiche Attribut der Tabelle `Student`. Diese Art des Vergleiches zeigt an, daß möglicherweise bei der Datenbankentwicklung eine Optimierung vorgenommen wurde (vgl. Kapitel 2.1.4) und ursprünglich folgende N:M-Beziehung vorlag:

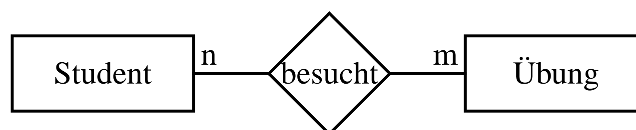


Abbildung 3.7 *Ursprüngliche Struktur der optimierten Beziehung*

Da auf diese Analysen im Kapitel 6 noch näher eingegangen wird, sollen hier keine weiteren Beispiele betrachtet werden. Anhand der drei analysierte Beispielanfragen ist aber bereits gut zu erkennen, wie und welche Informationen aus SQL-Anfragen abzuleiten sind. Da in realen Anwendungen eine weitaus größere Anzahl von Anfragen an die Datenbank gestellt wird, lassen sich auch noch genauere Schlüsse über Schlüsselkandidaten, Beziehungen und Optimierungsstrukturen ziehen.

3.5 Probleme des Reverse Engineerings

An dem gegebenen Beispielschema konnten die semantischen Informationen relativ einfach ermittelt werden. Dies ist bei umfangreicheren Legacy-Systemen nicht immer so, wie Premerlani und Blaha in [PB95] im „Katalog von beobachteten Eigenarten des relationalen Datenbankentwurfs“ zusammengestellt haben.

Ein Legacy-Datenbanksystem beinhaltet nach den Beobachtungen in [PB95] neben mehreren Implementierungsvarianten für die gleiche semantische Information auch im Laufe der Jahre unterschiedliche verwendete Standards und Funktionalitäten für Implementierung und Design. Dies sollte beim Reverse Engineering berücksichtigt werden. Zudem entstehen in Systemen, die über lange Zeit genutzt werden, häufig Datenstrukturen, die von den Anwendungen nicht mehr genutzt werden, weil diese Informationen z.B. durch Umstrukturierungen auf anderem Weg zu bekommen oder veraltet sind. Die ursprünglichen Datenstrukturen wurden aber nicht aus der Datenbank entfernt, da z.B. aufgrund mangelnder Dokumentation niemand genau wußte, ob noch Anwendungen existieren, die hierauf zugreifen. Auf diese Art und Weise kommt es in Legacy-Systemen auch häufiger vor, daß Datenstrukturen doppelt existieren, um eine „Abwärtskompatibilität“ zu gewährleisten. Die so entstehenden Redundanzen werden bewußt in Kauf genommen, da es im betrieblichen Alltag häufig nicht möglich ist, vor jeder Änderung oder Anpassung eine Analyse des Gesamtsystems durchzuführen, um so einen genauen Überblick zu bekommen. All dies ist auch eine Folge der mangelnden Dokumentation und zeigt die Notwendigkeit einer vollständigen und aktuellen Dokumentation vor allem während der Betriebsphase der Informationssysteme.

Zudem muß sich der Reverse Engineer darüber klar sein, daß alle Hinweise auf versteckte semantische Informationen im Schema, in den Ausprägungen und in den Anfragen einer relationalen Datenbankanwendung keine unumstößlichen Fakten sind. Auch in dem kleinen Beispiel wurde zunächst angenommen, daß es sich bei der Äquivalenzklasse {`Professor.Name`, `Mitarbeiter.Name`, `Student.Name`} um eine Fremdschlüsselbeziehung handelt, was durch die Analyse der Ausprägungen widerlegt wurde. Genauso kann es vorkommen, daß von falschen Voraussetzungen ausgegangen wird. So ist es z.B. möglich, daß einige Entwickler `SELECT`-Anfragen immer mit dem `DISTINCT`-Schlüsselwort nutzen, obwohl das in vielen Fällen überflüssig ist. Wenn dann der Reverse Engineer dies nicht berücksichtigt, wird er im Verlaufe seiner Analyse widersprüchliche Ergebnisse erhalten.

4 Cliché-Erkennung

Diese Arbeit befaßt sich mit der Cliché-Erkennung in relationalen Datenbankanwendungen. In diesem Kapitel soll zunächst der Begriff des Clichés definiert werden. Es werden anschließend verschiedene Möglichkeiten vorgestellt, Clichés in Quelltexten zu erkennen, bevor speziell auf Clichés aus dem Bereich relationaler Datenbankanwendungen eingegangen wird.

4.1 Software-Clichés

Im allgemeinen versteht man unter einem Cliché, einen „vielgebrauchten und daher nichtssagenden Ausdruck“ [BER90]. Als Software-Cliché hingegen bezeichnet man im Reengineering ein stereotypisches Programmfragment. Diese Definition hat im Vergleich mit der allgemeinen Definition also lediglich die Bedeutung eines vielgebrauchten Ausdrucks gemeinsam. Im Gegensatz zur allgemeinen Clichédefinition werden Software-Clichés (im folgenden als Cliché bezeichnet) eingesetzt, um Informationen über die untersuchte Software zu erhalten. Das Auffinden häufig genutzter Programmfragmente in Form von Algorithmen wie zum Beispiel Sortierverfahren gibt Auskunft über die dynamische Semantik einer Anwendung. Die Nutzung typischer Datenstrukturen wie zum Beispiel Listen oder Hash-Tabellen gibt Auskunft über den statischen Aufbau eines Programmes. Durch das Auffinden von solchen Clichés im Quelltext kann man von der eigentlichen Implementierung abstrahieren, so daß sich die Analyse der Semantik eines Programmes auf der höheren Abstraktionsebene vereinfacht.

Der Begriff des Clichés wurde 1990 von Linda Wills in [RW90] definiert. Sie geht davon aus, daß Programmierer immer wiederkehrende Programmieraufgaben auch immer auf die gleiche Art lösen und versucht so, die Semantik von LISP-Programmen vollkommen automatisch, nur anhand des Quellcodes herauszufinden (vgl. Kapitel 4.2.2).

Linda Wills unterscheidet fixe und variable Teile eines Clichés. Bei der in Kapitel 3 vorgestellten typischen `SELECT`-Anweisung, die das Schlüsselwort `DISTINCT` enthält (vgl. Abbildung 3.6) und zur Widerlegung der Schlüsseleigenschaft von Attributen eingesetzt wird, ist zum Beispiel die Benutzung der `DISTINCT`-Angabe ein fester Bestandteil des typischen Musters. Die selektierten Attribute oder der Aufbau der `WHERE`-Klausel der Anfrage hingegen können variieren.

4.2 Möglichkeiten zur Erkennung von Clichés

Das Ziel der Cliché-Erkennung kann es sein die dynamische Semantik eines Programmes zu analysieren oder Informationen über benutzte Datenstrukturen zu erhalten. Dazu wird das Wissen über typische Strukturen, die im Code genutzt werden können, als Clichés formuliert. Bei der Definition dieser Clichés stößt man aber auf folgende Schwierigkeiten:

(1) Nicht verwertbarer Code

Ein Programm ist nicht einfach eine Aneinanderreihung von Clichés. Dieser für die Cliché-Erkennung nicht verwertbare Code sollte auch nicht betrachtet werden.

(2) Streuung von Clichés

Ein Cliché besteht nicht notwendigerweise aus Codefragmenten, die direkt aufeinderfolgen. Vielmehr ist es durchaus üblich, daß Teile eines Clichés über den gesamten Code verteilt sind. Die Cliché-Erkennung sollte in der Lage sein, auch kleinste Teile, die Bestandteile eines Clichés sein könnten, zu erkennen und zu einem Cliché zusammenzusetzen.

(3) Verzahnung von Clichés

Es gibt typische Programmuster, die in verschiedenen Clichés wiederzufinden sind. Die Cliché-Erkennung sollte berücksichtigen, daß sich ein Cliché aus verschiedenen anderen Clichés zusammensetzen kann. Ebenso ist es möglich, daß sich mehrere Clichés in Teilen überlappen.

(4) Syntaktische Variationen

Viele Programmiersprachen bieten die Möglichkeit, die gleiche Semantik einer Anweisung durch unterschiedliche syntaktische Konstrukte zu erreichen (vgl. i+1, i++ usw. in C++). Ein Cliché sollte unabhängig von diesen Variationsmöglichkeiten erkannt werden. Es sollte von syntaktischen Variationen abstrahiert werden.

(5) Variationen in der Implementierung

Ein Problem läßt sich meist auf mehrere Arten lösen und implementieren, so kann man zum Beispiel die Suche eines bestimmten Elementes in einer Menge als Binärsuche oder lineare Suche implementieren. Es ist nicht möglich, für alle

denkbaren Arten der Implementierung eines typischen Algorithmus eigene Clichés zu definieren. Für die Cliché-Erkennung sollten konkrete Implementierungsvarianten keine Rolle spielen.

In [QUI94] stellt Alex Quilici heraus, daß alle Cliché-Erkennungsansätze einige Gemeinsamkeiten haben. Dabei wird der Quellcode zunächst in eine dedizierte Datenstruktur überführt, die den Daten- und Kontrollfluß des Programmes beschreibt. Daneben existiert eine Bibliothek, die Spezifikationen gängiger Clichés enthält. Aus den im Code enthaltenen Clichés wird dann meist eine baumartige Struktur abgeleitet, die die einzelnen Clichés generalisiert bzw. spezialisiert, so daß aus der Wurzel eines Baumes eine allgemeine Semantik des geparsten Programmes abzulesen ist.

Dabei unterscheidet Quilici Top-Down- und Bottum-Up-Ansätze. Die Top-Down-Ansätze verlangen zunächst einmal die Kenntnis über das generelle Ziel des Programmes, um dann im vorliegenden Quelltext nach Clichés zu suchen, die zur Erreichung des gesuchten Zieles geeignet erscheinen. Da aber häufig keine genauen Informationen über das jeweilige Programm vorliegen, sind diese Ansätze in der Praxis kaum einsetzbar. Die Bottum-Up-Ansätze beginnen mit dem Quellcode und bestimmen, welche Clichés hierin enthalten sind. Aus diesen meist recht speziellen Clichés wird auf globalere Clichés geschlossen, bis das Ziel eines Programmes erkannt ist oder keine passenden Clichés mehr gefunden werden. Diese Ansätze sind meist sehr ineffizient, da viele Codefragmente in mehreren Clichés enthalten sein können und jedes Cliché auch wieder Teil eines anderen, globaleren Clichés sein kann.

Die folgenden Absätzen beschreiben existierende Ansätze, mit Hilfe von Clichés Analysen der dynamischen Semantik von Programmen durchzuführen oder Rückschlüsse auf statische Datenstrukturen zu ziehen und wie die oben beschriebenen Probleme gelöst werden. Der grundsätzliche Unterschied zwischen den Text- und den Graphparsingansätzen besteht dabei in der Bedeutung der Zwischenstruktur. Während sich bei den Textparsingansätzen die aufzubauende Zwischenstruktur (zum Beispiel in Form von abstrakten Syntaxgraphen) eng an die Syntax des Quellcodes anlehnt, wird bei den Graphparsingansätzen durch den Aufbau von Kontroll- und Datenflußgraphen von der Syntax abstrahiert. Keiner der ausgewählten Ansätze befaßt sich jedoch mit der Erkennung von Clichés speziell im Bereich Datenbanken. Bisher existieren hier noch keine Cliché-Analysatoren.

4.2.1 Textparsing

Eine naheliegende Möglichkeit, Clichés in einem Programm zu entdecken, ist die Suche nach Mustern direkt im Programm. Diesen Ansatz verfolgen Santanu Paul und Atul Prakash in [PP94]. Sie definieren die Clichés (Pattern) in einer eigenen Sprache und suchen nach passenden Fragmenten (Matches) im Quelltext der Programme. Die Patternsprache ist eng an eine Programmiersprache angelehnt (hier C / C++) und umfaßt neben den Elementen der Programmiersprache noch einige Erweiterungen mit speziellen Symbolen zur Mustererkennung.

Pattern

```
$f_1 = '*max*'
%%
$t_1 $f_1($*v)
$*d
{*
   @[while|dowhile|for]
   {*
      if ($v_2[#] > $v_3)
          $v_3 = $v_2[#]
   *}
*}
```

Match

```
int find_max (int_arr, N)
int int_arr[];
int N;
{ int i;
  int maxstore;
  maxstore = int_arr[0];
  for (i=1; i<N; i++)
  {
     if (int_arr[i] > maxstore)
         maxstore = int_arr[i];
  }
  return (maxstore);
}
```

Abbildung 4.1 *Pattern und passender Match*

In der Abbildung 4.1 ist auf der linken Seite ein Pattern in der von Paul/Prakash definierten Sprache zu sehen. Es beschreibt die Suche nach dem Maximum in einem Array. Ein dazu passender Match ist in dem Auszug aus einem C-Programm auf der rechten Seite zu sehen. Die folgende Tabelle ausgesuchter Patternsymbole soll dem Leser einen kleinen Überblick verschaffen. Eine ausführliche Erläuterung kann [PP94] entnommen werden.

Syntaktische Einheit	Pattern Symbol
Deklaration (-enmenge)	$d ($*d)
Typ	$t
Variable (-nmenge)	$v ($*v)
Funktion	$f
Ausdruck (-smenge)	# (#*)
Anweisung (-sfolge)	@ (@*)

TABELLE 1.

Diese Patternsprache wird im Pattern Matching System SCRUPLE [PP94] genutzt, das in einem gegebenen Programmcode nach passenden Fragmenten für die beschriebenen Pattern sucht. Die für das Pattern Matching relevanten Bausteine der SCRUPLE-Architektur sind in der folgenden Abbildung zu sehen.

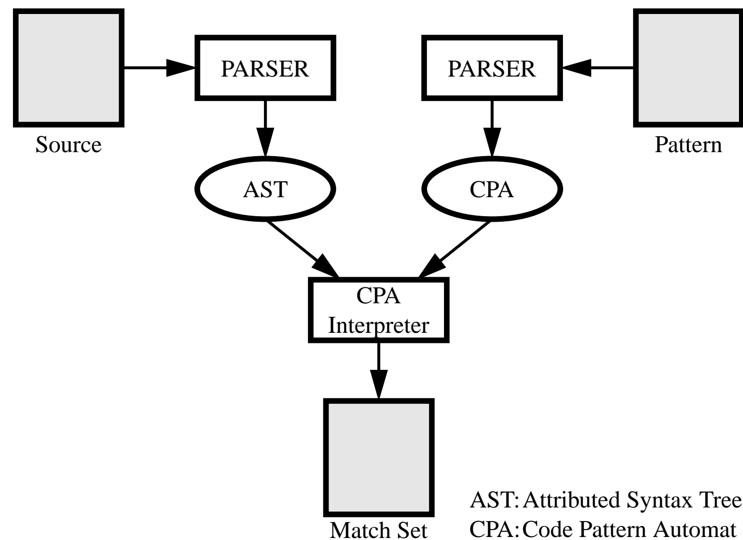


Abbildung 4.2 *Das Pattern Matching System (vereinfacht)*

Der Quellcode des zu untersuchenden Programms wird zunächst mit Hilfe eines Parsers in einen attributierten Syntaxbaum (AST) überführt. Ein anderer Parser formt das gesuchte Pattern in einen Code Pattern Automaten (CPA) um. Die eigentliche Suche nach Pattern im Quelltext vollzieht dann der CPA-Interpreter. Dieser führt den Code Pattern Automaten mit dem attributierten Syntaxgraphen des Quellcodes als Eingabe aus.

Bei dem attributierten Syntaxbaum handelt es sich um eine Datenstruktur, die die abstrakte Syntax des Quellcodes widerspiegelt. Die inneren Knoten des AST stellen die Terminale und Nichtterminale des Programmes dar, die durch Attribute in den Blättern näher beschrieben werden. Solche Terminale bzw. Nichtterminale können Funktionen, Deklarationen, Anweisungen, Ausdrücke oder andere Terminal- und Nichtterminalsymbole der zugrundeliegenden Programmiersprache sein.

Den AST für den folgenden Beispielcode zeigt Abbildung 4.3.

```
{
  while (z>1) z=z-2;
  y=0;
  p=0;
  if (t==1) x=1;
}
```

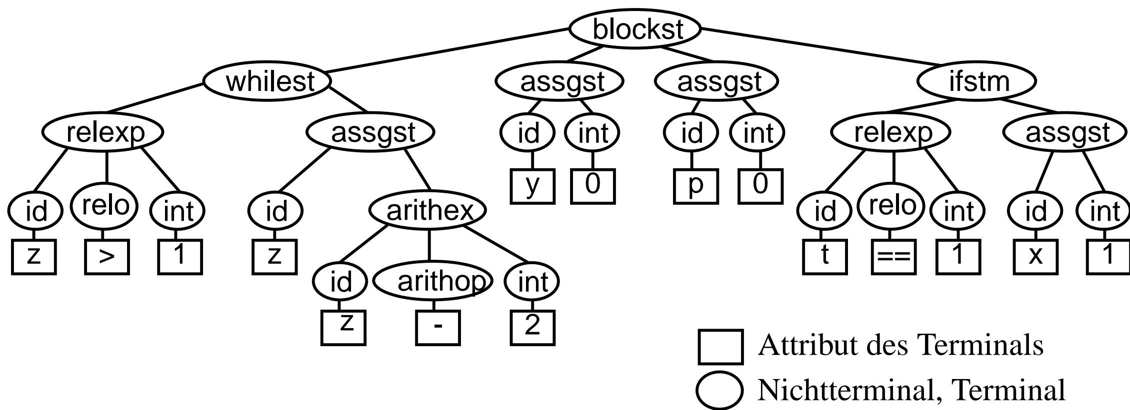


Abbildung 4.3 *AST für den Beispielcode*

Die oben genutzten Bezeichner und deren Bedeutung zeigt folgende Tabelle:

Bezeichner	Bedeutung
blockst	Anweisungsblock
whilest	While-Schleife
assgst	Zuweisung
ifstm	If-Anweisung
relexp	Vergleichsausdruck
arithex	Arithmetischer Ausdruck
relo	Vergleichsoperator
arithop	Arithmetischer Operator
id	Bezeichner

TABELLE 2.

Bei dem Code Pattern Automaten handelt es sich um einen nichtdeterministischen endlichen Automaten (siehe Abbildung 4.4). Dabei wird der Nichtdeterminismus des Automaten benötigt, um spezielle Patternsymbole wie @*, #* aufzulösen. Der oben

beschriebene AST ist die Eingabe für den CPA. Da normale endliche Automaten aber eine solche Baumstruktur nicht verarbeiten können, muß der CPA erweitert werden.

Definition 4.1

$\mathrm{CPA} = \langle Q, \Sigma, A, \Gamma, q_0, F \rangle$ *mit:*

- *Q ist eine Menge von Zuständen*
- *Σ ist das Eingabealphabet, das aus Knoten des AST besteht, die die syntaktischen Elemente darstellen. Dabei ist $\Sigma = \Sigma_N \cup \Sigma_L$. Σ_N beschreibt die internen Knoten des AST und Σ_L beschreibt die Blätter.*
- *A ist eine Menge von Navigationsfunktion im AST mit A = {moveto_leftchild, moveto_rightsibling, moveto_parent}. Die Semantik der einzelnen Funktionen sei dem Namen entnommen. Die Funktionen werden vom CPA Interpreter benutzt, um den AST zu durchlaufen.*
- *Γ ist eine Menge von Transitionsfunktionen mit $\Gamma = \{\mathrm{CAT}, \mathrm{VAL}\}$ mit*
  - *CAT: $Q \times \Sigma_N \rightarrow 2^{Q \times A^{\dagger}}$ mit Label:*

    CAT <category>; ACTION: <actions>

    Die CAT-Kante spezifiziert die syntaktische Kategorie zu der der Eingabeknoten gehören muß. Falls zum Beispiel ein `while_stmt` *gefunden wird, kann auch eine Kante mit dem label CAT<stmt> durchlaufen werden.*

    $2^{Q \times A^{\dagger}}$ *drückt den Nichtdeterminismus der Transitionen aus, d.h. es gibt mehrere gültige Transitionen für einen bestimmten Zustand. Es werden alle möglichen Folgezustände berücksichtigt.*
  - *VAL: $Q \times \Sigma_L \rightarrow 2^{Q \times A^{\dagger}}$ mit Label:*

    VAL <value>; ACTION: <actions>

    Die VAL-Kante kann durchlaufen werden, falls der Attributwert des Blattknotens dem Wert entspricht der durch <value> angegeben wird.
- *q_0 in Q ist der Anfangszustand.*
- *$F \subseteq Q$ ist eine Menge von möglichen Endzuständen.*

Die folgende Abbildung zeigt ein Beispielpattern und den daraus generierten CPA:

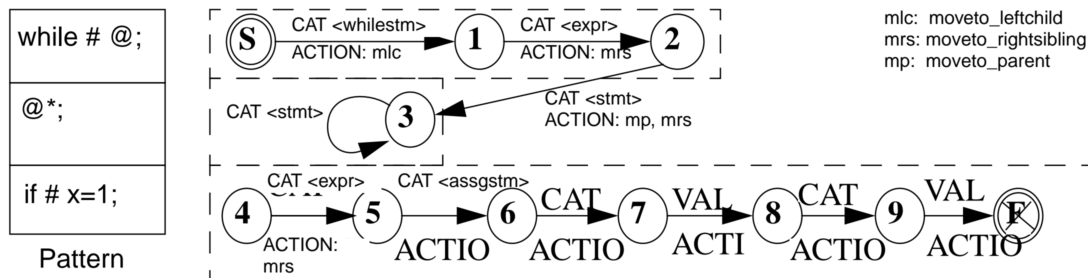


Abbildung 4.4 *Code Pattern Automat zu Beispielpattern*

Der CPA-Interpreter simuliert den CPA an einem AST und erzeugt einen *MatchSet*, d.h. eine Menge aller Codefragmente, die dem Eingabepattern entsprechen. Der Interpreter simuliert die notwendigen Zustandsüberführungen und setzt den Zeiger, der auf den aktuellen Knoten verweist, auf den nächsten Eingabeknoten. Dieser wird durch die Aktionen der Transitionspfeile bestimmt. Ein Match wird schließlich gefunden, wenn der CPA den Endzustand erreicht.

Wie oben bereits erwähnt ist ein syntaktischer Ansatz zur Cliché-Erkennung zunächst naheliegend. Allerdings ergeben sich durch die stark an eine bestimmte Programmiersprache angelehnte Definition der Pattern Probleme, die einem praktischen Einsatz im Wege stehen. Falls beispielsweise mehrere Programmiersprachen parallel eingesetzt werden, müßte für jede Sprache eine eigene Patternsprache und die Umwandlung in den CPA entwickelt werden. Außerdem gibt es keine Lösung für das Problem der syntaktischen Variationen, so daß alle möglichen syntaktischen Varianten eines Clichés spezifiziert werden müßten, was in der Praxis natürlich nicht machbar ist. Die Laufzeit dieses Clichéparsers geben Paul und Prakash mit O(N^2) in Abhängigkeit der Knotenanzahl des AST an, so daß sie mit der Laufzeit von Algorithmen zur Erkennung von regulären Ausdrücken in Strings zu vergleichen ist [PP94].

4.2.2 Graphparsing

Anstatt Clichés direkt im Quelltext zu suchen, wird der Kontrollfluß der Programme bei Graphparsingansätzen in eine sprachunabhängige, graphische Notation umgewandelt, um so das Problem der Variationen zu umgehen.

Dabei werden für die eigentliche Analyse sog. *Graphgrammatiken* eingesetzt. In [SCH87] wird ein Graph folgendermaßen definiert. „Ein gerichteter, knoten- und kantenmarkierter Graph über dem endlichen Knotenmarkierungsalphabet Σ_V und dem endlichen Kantenmarkierungsalphabet Σ_E ist das Tripel g=(V, E, L) mit:

a) $V \subseteq N^*$ ist eine endliche Menge. Elemente von V heißen Knoten oder Knotenbezeichner von g.
b) $E \subseteq V \times \Sigma_E \times V$ ist eine endliche Menge von Kanten. $(v, x, w) \in E$ heißt x-Kante von v nach w.
c) $L: V \to \Sigma_v$ ist die Knotenmarkierungsfunktion. Wenn L(v)=A, dann nennen wir v eine A-Knoten.“ [SCH87]

In [NAG79] werden dann Graphgrammatiken wie folgt beschrieben: „In einem Graphen d wird die linke Seite d_l einer Graph-Regel aufgefunden. Sie ist zu ersetzen durch die rechte Seite d_r dieser Regel, wobei i.a. ein veränderter Graph d’ entsteht“ [NAG79]. Eine Graph-Regel hat „drei Komponenten p=(d_l, d_r, E), die linke bzw. rechte Seite d_l bzw. d_r und eine *Einbettungsüberführung* E“ [NAG79]. Dabei beschreibt die Einbettungsüberführung, „wie die neu eingesetzte rechte Seite in den Wirtsgraphen, aus dem die linke Seite herausgenommen wurde, eingehängt werden soll“ [NAG79].

4.2.2.1 Der Recognizer von Linda Wills

Das Überführen des Quellcodes in eine graphische Zwischenstruktur hat auch Linda Wills gewählt, die mit ihrem „Recognizer“ versucht, die Semantik von LISP-Programmen offenzulegen [RW90]. Dabei geht sie davon aus, daß jedes Cliché, wie zum Beispiel Sortierverfahren, Hash-Tabellen usw. einen charakteristischen Kontrollfluß besitzt. Dies nutzt sie aus, indem sie denkbare Clichés mit ihren charakteristischen Kontrollflüssen in einer sog. Cliché-Bibliothek ablegt und die zu untersuchenden Programme mit Hilfe der Bibliothek untersucht. Dazu werden die Programme zunächst in eine graphische Notation (Plan Calculus) [RW90] überführt und dieser dann in ein Flußdiagramm umgewandelt. Anhand der Cliché-Bibliothek und den dazu gehörenden Graphgrammatiken wird dann analysiert, welche typischen Kontrollflüsse in dem Diagramm des zu untersuchenden Programmes enthalten sind. Das Ergebnis dieses

Schrittes ist eine baumartige Datenstruktur, der sog. Design Tree, aus dem dann eine Dokumentation generiert werden kann, die die dynamische Semantik des ursprünglichen Programmes widerspiegelt.

Anhand des folgenden LISP-Programmes, soll die Vorgehensweise des Recognizers näher erläutert werden.

```
(DEFUN Table_Lookup (Table Key)
   (LET (( Bucket (ARef Table ( Hash Key Table))))
      (LOOP
         (IF (NULL Bucket) (RETURN Nil))
         (LET (Entry (CAR Bucket))
            (IF (EQUAL (KEY Entry) Key) (RETURN Entry)))
        (SETQ Bucket (CDR Bucket)))))
```

Hierbei handelt es sich um ein Programm, das nach einem bestimmten Eintrag in einer Liste sucht. Die Liste ist als Hash-Tabelle implementiert. Der Funktion `Table_Lookup` werden eine Tabelle `Table` und ein zu suchender Schlüssel `Key` übergeben. Mit der Funktion `Hash` wird die Position in `Table` errechnet, an der ein Eintrag mit dem Schlüssel `Key` abgelegt ist. `ARef` gibt alle Einträge an der entsprechenden Position zurück, die in `Bucket` gespeichert werden. Diese Liste wird dann in der nachfolgenden Schleife dursucht. Falls `Bucket` leer ist, bricht `Table_Lookup` mit der Rückgabe `Nil` ab. Der Variablen `Entry` wird das erste Element der Liste `Bucket` zugewiesen (mittels head-Funktion `CAR`) und, falls es mit dem gesuchten Schlüssel übereinstimmt, zurückgegeben. Sonst wird `Bucket` auf den Rest der Liste gesetzt (mittels tail-Funktion `CDR`) und mit dieser Liste weitergesucht, bis das gesuchte Element gefunden wird oder die Liste komplett durchsucht ist.

Die zu analysierenden Programme werden zunächst in den Plan Calculus überführt. Das ist eine graphische Datenstruktur, die von den Variationen in der Syntax eines Algorithmus abstrahieren soll. Der Plan Calculus wird repräsentiert durch Rechtecke, die Operationen und Tests darstellen, und Pfeilen, die den Kontroll- und Datenfluß veranschaulichen. Abbildung 4.5 zeigt auf der linken Seite den Plan Calculus für das Beispielprogramm. Es wurde eine Folge von 3 Clichéoperationen gefunden:

(1) `hash` *berechnet die Tabellenposition für den Eingabewert.*

(2) `select` *gibt die Menge aller Einträge an einer bestimmten Position in der Tabelle zurück.*

(3) `retrieve` *beschreibt die Suche nach dem gewünschten Element in einer Menge von Tabelleneinträgen.*

Anhand der Pfeile kann man den Datenfluß nachvollziehen. So gibt zum Beispiel `select` die Einträge der Tabelle, die auch `hash` als Eingabe übergeben wird, zurück. Die Tabellenposition ist diejenige, die von `hash` berechnet wurde. Die Beziehung zwischen `select` und `retrieve` lassen sich auf diese Weise ebenfalls ablesen.

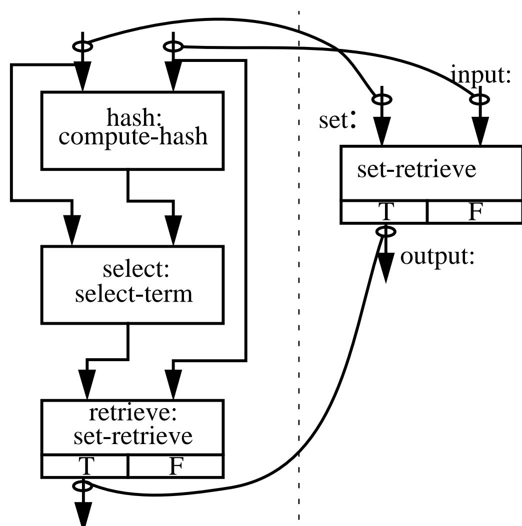


Abbildung 4.5 *Overlay für Hash-Table Suche*

Die rechte Seite der Abbildung zeigt einen weiteren Plan Calculus namens `set-retrieve`. Wenn eine solche Operation erfolgreich ist, gibt sie ein Element der Eingabemenge zurück, dessen Schlüssel mit dem Eingabeschlüssel übereinstimmt. Diese Vor- und Nachbedingungen können separat in einer speziellen logischen Sprache beschrieben werden. Die oben erwähnte Cliché-Bibliothek enthält nun für viele typische Programmstrukturen Spezifikationen in dieser Art, d.h. eine Plan Calculus-Darstellung und die dazugehörenden Bedingungen. Die Beziehung zwischen einer möglichen Implementierung und der dazugehörenden Spezifikation wird über sogenannte *Overlays* realisiert. Ein Overlay definiert eine Abbildung zwischen Instanzen eines Plans zu Instanzen eines anderen Plans (in der Abbildung dargestellt durch die Linien mit den kreisförmigen Enden). In der Cliché-Bibliothek können nun mehrere Overlays definiert sein, die sich auf einen Plan beziehen. So kann ein Cliché, wie hier die Suche nach einem Element in einer Menge (`set-retrieve`), auf verschiedene Arten implementiert werden. Der oben beschriebene `hash-table-retrieve` ist nur ein mögliche Implementierung. In der Cliché-Bibliothek muß für jede mögliche Implementierungsvariante eines Clichés ein Overlay zur Spezifikation des Clichés vorhanden sein.

Die Erkennung der Overlays wird mit Hilfe von kontextfreien Graphgrammatikregeln[1] durchgeführt. Der Plan Calculus wird hierzu in einen Flußgraphen überführt. Dabei werden die Rechtecke des Plans zu Knoten des Flußgraphen. Der Datenfluß wird als

Kanten zwischen den Knoten dargestellt. Andere Informationen wie Vor- und Nachbedingungen werden als Attribute des Flußgraphen dargestellt.

Die Cliché-Erkennung sucht nach Teilgraphen und ersetzt diese durch Graphen, die einer höheren Abstraktionsebene entsprechen. Abbildung 4.6 zeigt einen Teil der Regeln für das Beispiel.

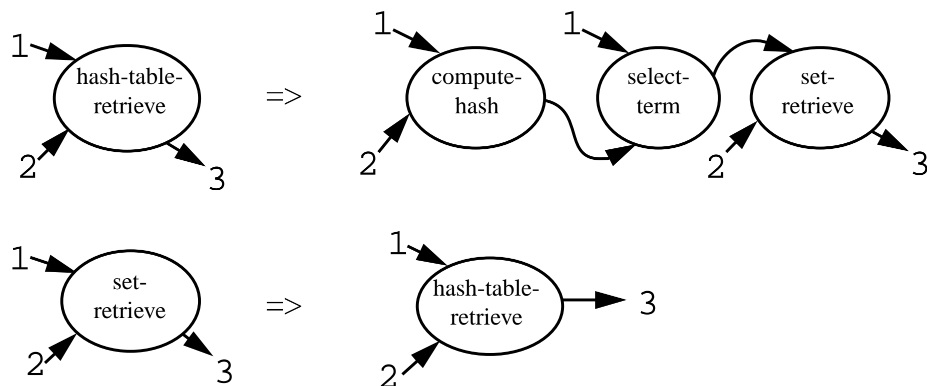


Abbildung 4.6 *Auszug aus der Graphgrammatik zur Hash-Table-Suche*

Die obere Regel beschreibt die Erkennung eines `hash-table-retrieve`, wie er im Beispiel implementiert wurde. Falls der Recognizer diese Regel im Verlaufe seines Parservorganges benutzt, bedeutet das, daß ein `hash-table-retrieve`-Cliché erkannt wurde. Der Overlay zum `set-retrieve` wird dann mit Hilfe der unteren Regel erkannt. Der Recognizer findet somit heraus, daß das Problem, ein Element aus einer Menge zu suchen `(set-retrieve)`, durch die Suche in einer Hashtabelle `(hash-table-retrieve)` implementiert wurde. Diese Informationen werden dann bei der Ableitung des Design-Tree benutzt.

1. Kontextfreie Graphgrammatikregel enthalten auf der linken Seite einen einzigen, nichtterminalen Knoten. [RW90]

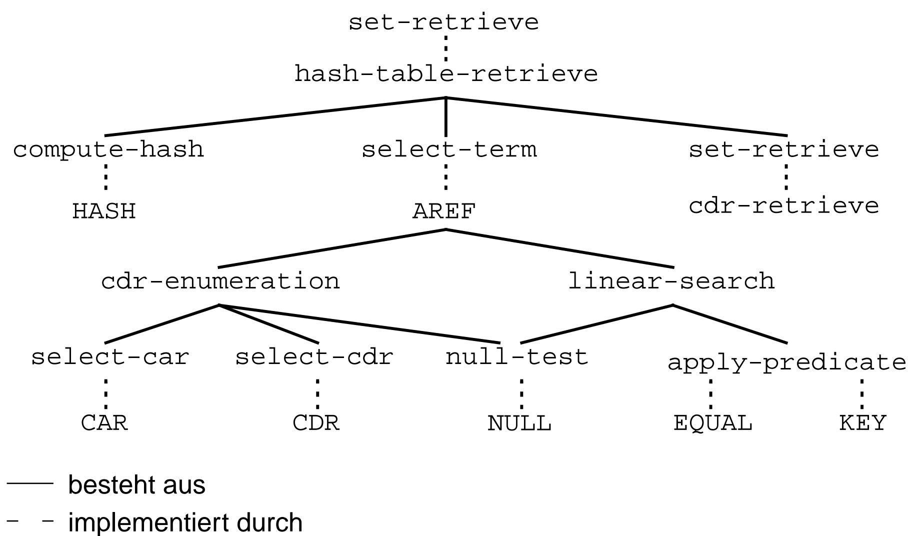


Abbildung 4.7 *Design-Tree des Beispielprogrammes*

Abbildung 4.7 zeigt den Baum für das Beispielprogramm aus dem eine Dokumentation über die Semantik des Programmes generiert wird. Der Wurzel des Baumes kann man das allgemeinste Cliché, d.h. das Cliché der höchsten Abstraktionsebene, entnehmen, welches erkannt wurde. Die gestrichelten Linien geben Auskunft über die gewählte Implementierungsvariante dieses Clichés, die sich eventuell wieder aus einem oder mehreren Clichés zusammensetzt. Die Blätter des Baumes enthalten die im Programm benutzten Bezeichner und Funktionen.

Dieser Ansatz zur Cliché-Erkennung bietet eine Möglichkeit, die Abhängigkeit zu einer bestimmten Programmiersprache zu umgehen, indem das zu untersuchende Programm zunächst in eine graphische Notation überführt wird. Für alle Programmiersprachen, die man untersuchen möchte, ist dann lediglich eine Überführung der konkreten Sprache in die graphische Notation bereitzustellen. Da sich diese graphische Notation zudem auf die Darstellung des Kontrollflusses beschränkt, spielen syntaktische Variationen im Code keine Rolle, da sie durch den gleichen Kontrollfluß repräsentiert werden. Allerdings führt dies zum Verlust der Informationen über den syntaktischen Aufbau eines Algorithmus. In [RW90] wird darauf hingewiesen, daß diese Vorgehensweise für kommerzielle Programme nicht geeignet ist, da die Suche in der Clichébibliothek nach allen möglichen Ableitungen des Eingabegraphen zu ineffizient ist. Deshalb wird hier zum Beispiel vorgeschlagen, Heuristiken einzusetzen, um den Ableitungsvorgang abzukürzen.

4.2.2.2 Der Hybridansatz von Alex Quilici

Mit seinem Hybridansatz versucht Quilici den menschlichen Prozeß nachzubilden, den Programmierer wählen, wenn sie versuchen ein ihnen unbekanntes Programm zu verstehen. Dabei fand er heraus, daß das Auffinden bestimmter Schlüsselworte im Code die Wahl der betrachteten Clichés beeinflußt. So geht man davon aus, daß ein `while`, welches einen Test mit einer Variablen und einem Vergleich auf größer oder kleiner enthält, eine Iteration über einen Bereich bedeutet. Wenn dann ein erster Ansatzpunkt im Code gefunden wird, versucht man diesen genauer zu spezifizieren. Erst wenn die Bedeutung des zugehörigen Codes bekannt ist, wird nach weiteren Schlüsselworten gesucht. Das Auffinden bestimmter Clichés im Code veranlaßt den Programmierer zudem, implizit Rückschlüsse über hieraus ableitbare Clichés zu ziehen, ohne explizit im Code danach zu suchen.

Aus diesen Beobachtungen wurde die Forderung abgleitet, daß eine Clichébibliothek indiziert und hierarchisch organisiert sein sollte, um die Anzahl der zu betrachtenden Clichés zu begrenzen. Dies soll durch die folgenden drei Informationen erreicht werden, die jedes Cliché enthält:

(1) *Indexbeschränkungen*

Jede Spezifikation in der Clichébibliothek enthält Informationen über Anweisungen, Programmfragmente oder andere Clichés, die im Code vorhanden sein müssen, damit das jeweilige Cliché überhaupt erfüllt ist. Das Auffinden eines Teils einer solchen Indexinformation löst dann die Überprüfung aller anderen für den Index des Clichés benötigten Programmfragmente aus.

(2) *Spezialisierungsbeschränkungen*

Das Auffinden einer Indexkomponente und die Validierung des gesamten Index führt meist dazu, das zunächst relativ generelle Clichés erkannt werden. Deshalb sollte für jedes Cliché vermerkt werden, welche spezielleren Clichés im Code zu suchen und zu validieren sind.

(3) *Implikationsbeschränkungen*

Durch das Auffinden eines bestimmten Clichés im Code kann auf Programmfragmente geschlossen werden, die ebenfalls im Code enthalten sind. Eine Suche nach diesen implizierten Clichés kann somit entfallen. Jedes Cliché muß deshalb explizit alle Clichés angeben, die auch im Code enthalten sind, wenn dieses Cliché erkannt wurde.

Zur eigentlichen Cliché-Erkennung wird dann das zu untersuchende Programm in einen abstrakten Syntaxgraphen überführt, aus dem man alle Komponenten und ihre Beziehungen zu anderen Komponenten ablesen kann. Dabei kann es sich bei einer Komponente sowohl um eine einfache Anweisung der zugrundeliegenden Programmiersprache handeln als auch um abstraktere Konstrukte wie beispielsweise Schleifen.

Der Algorithmus zur Cliché-Erkennung durchsucht dann nacheinander alle Komponenten. Dabei wird jede einzelne Komponente zunächst soweit wie möglich spezialisiert, d.h. es wird anhand der Spezialisierungsinformationen getestet, welche Spezifikation vorliegt und wie weitere Spezifikationen aussehen könnten. Falls keine weitere Spezifikation der Ausgangskomponente möglich ist, wird überprüft, welche Clichés durch die Ausgangskomponente oder jede einzelne Spezifikation indiziert werden. Für alle indizierten Clichés wird dann zunächst geprüft, ob die hierfür benötigten Schlüsselanweisungen im zu untersuchenden Quellcode vorliegen. Nach erfolgreichem Indextest wird dann versucht, das gesamte Cliché im Code zu verifizieren. Die entsprechende Komponente im Code wird markiert und alle hierdurch implizit erkannten Clichés werden ebenfalls markiert. Wenn dann alle Komponenten des Quelltext durchsucht wurden, kann man anhand der Liste erkannter Clichés auf die Semantik des untersuchten Programmes schließen.

Der von Quilici entwickelte Bottum-Up/Top-Down-Ansatz basiert auf einer indizierten, hierarchisch organisierten Clichébibliothek. Es wird zunächst wie bei allen Bottum-Up-Ansätzen nach einem Cliché im Quellcode gesucht, das dann anhand der zusätzlich in der Bibliothek enthaltenen Informationen weiter spezialisiert wird. Die Spezialisierung wird dann in einem Top-Down-Schritt anhand des Quellcodes validiert. Durch diese Vorgehensweise wird nach Auffassung von Quilici die Anzahl der zu untersuchenden Clichés während der Clichéerkunng und somit deren Aufwand erheblich reduziert. Die Effizienz dieses Ansatzes hängt jedoch wesentlich von der Möglichkeit ab, die Indizes für die einzelnen Clichés effizient zu bestimmen.

4.3 Clichés im Bereich relationaler Datenbanken

Diese Arbeit befaßt sich mit der Erkennung von Clichés im Bereich von relationalen Datenbankanwendungen und wird zur Semantikanalyse von relationalen Datenbankschemata eingesetzt. Auch hier versteht man unter einem Cliché ein typisches Muster im Programm. Dabei lassen sich Clichés im Bereich relationaler Datenbanken in folgende Kategorien aufteilen. Beispiele zu den einzelnen Kategorien sind im Kapitel 6 zu finden:

(1) Clichés zur Erkennung von intrarelationalen Abhängigkeiten

Diese Clichés geben Aufschluß über Schlüsseleigenschaften oder funktionale Beziehungen einzelner Attribute einer Relation. Sie können natürlich auch Indikatoren gegen solche Eigenschaften sein.

(2) Clichés zur Erkennung von interrelationalen Abhängigkeiten

Bei dieser Art von Clichés werden Hinweise für oder gegen Fremdschlüsselbeziehungen zwischen Attributen *verschiedener* Relationen gesucht.

(3) Clichés zur Erkennung von Optimierungsstrukturen

Um einen möglichst effizienten Zugriff auf relationale Datenbanken zu erreichen, wird beim Datenbankentwurf für häufig zusammen genutzte Strukturen ein vereinfachter Zugriff zum Beispiel durch das Zusammenfassen von Attributen ermöglicht. Für das Auffinden dieser Strukturen können ebenfalls Clichés definiert werden.

Die Eingrenzung des Anwendungsbereiches auf relationale Datenbanken bringt einige Besonderheiten mit sich. So erfolgt der Zugriff auf relationale Datenbanken standardmäßig über SQL-Anfragen. Andererseits gestaltet sich eine Ausweitung der Cliché-Erkennung auf verschiedene Datenbanktypen schwierig, da sich viele Clichés, die zum Beispiel bei relationalen Datenbanken auftreten nicht auf andere Datenbanken

übertragen lassen. Clichés zur Erkennung von Fremdschlüsselbeziehungen sind zum Beispiel nicht auf Netzwerkdatenbanken übertragbar, da hier solche Beziehungen nicht existieren.

4.4 Cliché-Erkennung in relationalen Datenbankanwendungen

Die Eingrenzung des Anwendungsbereiches der Cliché-Erkennung auf relationalen Datenbankanwendungen bringt einige Besonderheiten mit sich. So wird zum Beispiel eine Sprachunabhängigkeit der Cliché-Erkennung nicht benötigt, da mit SQL ein allgemeingültiger Standard für relationale Datenbankanwendungen existiert. Zudem wird im Rahmen dieser Arbeit die dynamische Semantik der SQL-Anfragen nicht betrachtet, da zur semantischen Anreicherung des Datenbankschemas nur die genutzten statischen Datenstrukturen interessant sind. Es wird also nicht verfolgt, welches Ergebnis die untersuchten Anfragen liefern, sondern nur, welche Hinweise auf Schlüsselkandidaten, Fremdschlüsselbeziehungen, Optimierungsstrukturen etc. implizit in den Anfragen verwendet werden. Hieraus lassen sich dann Rückschlüsse auf das konzeptionelle relationale Datenbankschema ziehen (siehe Kapitel 3.4).

Eine weitere Vereinfachung gegenüber der allgemeinen Cliché-Erkennung ergibt sich daraus, daß die zu erkennenden Clichés im Bereich der statischen Analyse von SQL-Anfragen nicht allzu komplex werden. Die Anfragen haben eine fest definierte Syntax. Es kommen keine komplizierten Algorithmen zur Anwendung. All dies schränkt die Anzahl denkbarer Clichés ein, so daß eine effiziente Erkennung der Clichés möglich ist.

Allerdings ist es gerade durch die fest definierte Syntax manchmal wichtig, den genauen syntaktischen Aufbau einer SQL-Anfrage nachvollziehen zu können. Bei vielen denkbaren Clichés ist dies aber überflüssig. Dort führt eine mögliche Abstraktion vom syntaktischen Aufbau einer Anfrage sogar zu einer wesentlichen Vereinfachung.

5 Konzeption und Realisierung des Clichéparsers

In Kapitel 4 wurden existierende Ansätze zur Cliché-Erkennung vorgestellt. Dabei befaßte sich jedoch keiner der Ansätze speziell mit den Anforderungen der Cliché-Erkennung im Bereich relationaler Datenbankanwendungen. Es soll deshalb hier eine Möglichkeit der Cliché-Erkennung aufgezeigt werden, die speziell für diesen Bereich anwendbar ist. Dazu stellt Kapitel 5.1 die Konzeption des Clichéparsers vor, der im Rahmen dieser Arbeit entstanden ist. In Kapitel 5.2 werden dann geschichtete Graphgrammatiken eingeführt, die zur formalen Spezifiation der Clichés verwendet werden. Schließlich beschreibt das Kapitel 5.3 die Realisierung des Clichéparsers.

5.1 Konzeption

Alle in Kapitel 4 vorgestellten Ansätze zur Cliché-Erkennung weisen einige Gemeinsamkeiten auf. So werden die möglichen Clichés zunächst in einer Clichébibliothek spezifiziert. Der zu untersuchende Quellcode wird in eine spezielle, dedizierte Struktur überführt, die den Daten- und Kontrollfluß des Programmes beschreibt. Im eigentlichen Cliché-Erkennungsvorgang (mit Hilfe des *Clichéerkenners*) wird dann diese Struktur auf unterschiedliche Weise nach den spezifizierten Clichés durchsucht. Der hier vorgestellte Clichéparser orientiert sich an dieser Vorgehensweise. Er hat somit die Struktur der existierenden Cliché-Erkennungswerkzeuge. Die Vorgehensweise des Clichéparsers wird in der Abbildung 5.1 grob skizziert.

Da mit SQL eine genormte Datenbanksprache zur Verfügung steht, die von allen namhaften Datenbankherstellern unterstützt wird, soll sich der Clichéparser auf die Betrachtung der SQL-Datenbankzugriffe beschränken. Diese werden aus den Quelldateien der Applikation als (in einer Wirtssprache) eingebettetes SQL ausgelesen und in einen *abstrakten Syntaxgraphen* überführt, der die Struktur der SQL-Anweisungen wiedergibt. Die Knoten und Kanten eines abstrakten Syntaxgraphen entsprechen den zugrundeliegenden Sprachkonstrukten [RS97]. Die Spezifikationen der möglichen Clichés sind in einer Clichébibliothek zusammengefaßt. Der Clichéerkenner sucht in den abstrakten Syntaxgraphen der SQL-Anweisungen nach diesen typischen Mustern. Die

erkannten Clichés werden dann weiteren Analysewerkzeugen (vgl. [HEI98], [STR98]) als Textdatei zur Verfügung gestellt.

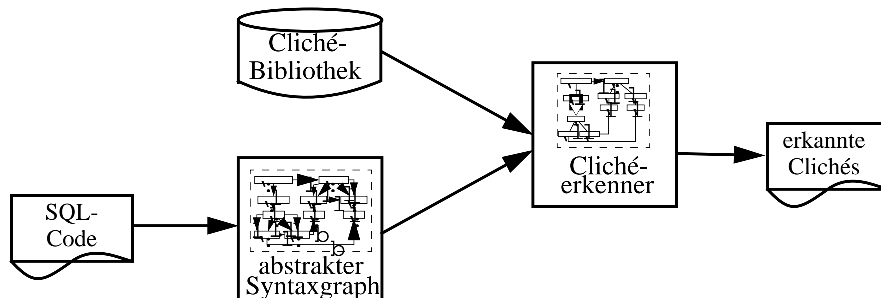


Abbildung 5.1 *Ablauf des Cliché-Erkennungsvorgangs*

Aus diesem Ablauf ergibt sich die Notwendigkeit einer genaueren Spezifikation

(1) der Struktur des abstrakten Syntaxgraphen und der Überführung der SQL-Anweisungen in diese Struktur.

(2) der Clichés und des Aufbaus der Clichébibliothek.

(3) des Clichéerkenners, der die spezifizierten Clichés in dem abstrakten Syntaxgraphen erkennt.

(4) des Formates der erkannten Clichés, die dann in weiteren Analysen genutzt werden.

Die Überführung der SQL-Anweisungen in den abstrakten Syntaxgraphen

Für die Clichéerkunng im Bereich relationaler Datenbankanwendungen bildet die Struktur und Syntax von SQL-Anfragen den Ausgangspunkt der Untersuchungen. In Kapitel 2.2 wurde bereits die Syntax der `SELECT`-Anweisung von SQL vorgestellt (vgl. Anhang A, [DAT89]). Die Struktur der SQL-Anweisungen ist als abstrakter Syntaxgraph darzustellen. Dies soll formal mit Progres (PROgrammierte GRaphErsetzungsSysteme) geschehen. Progres erlaubt die Spezifikation von Graphgrammatiken bzw. Graphersetzungssystemen, aus denen dann ausführbare C-Programme generiert werden können. Einen Überblick über Progres geben [SCH96a] und [SCH96b].

Die folgende Abbildung zeigt die für die abstrakten Syntaxgraphen benutzten Knotenklassen und deren Beziehungen als OMT-Objektdiagramm:

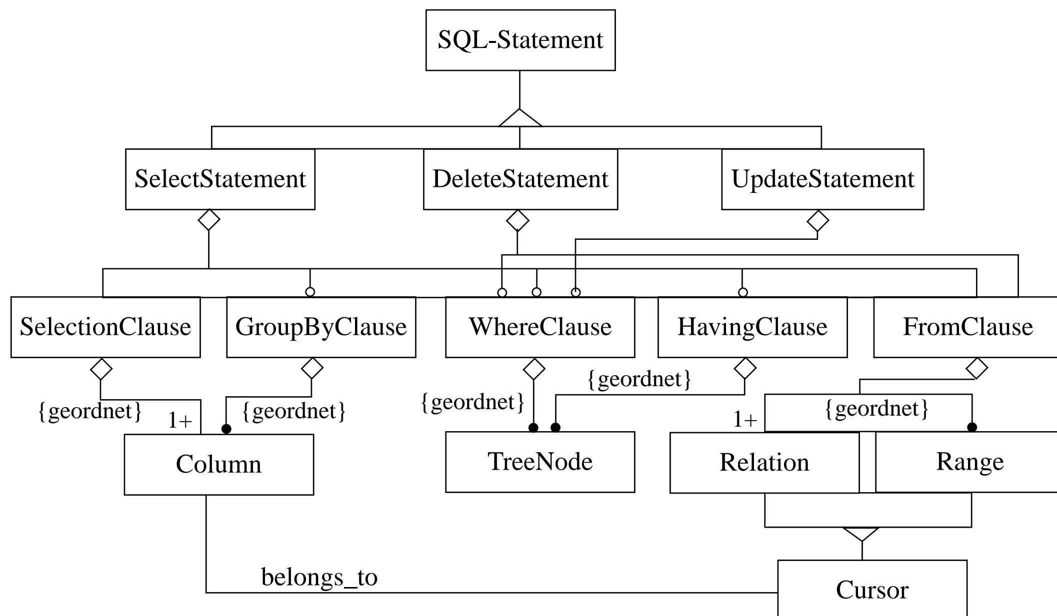


Abbildung 5.2 *Knotenklassen des abstrakten Syntaxgraphen*

Die Cliché-Erkennung soll sich im Rahmen dieser Arbeit zunächst auf die `SELECT`-Anweisungen beschränken, da dies die am häufigsten genutzte Anweisungen der DML ist. Die Betrachtung der `INSERT`, `UPDATE` und `DELETE`-Anweisung wird im weiteren vernachlässigt. Wie an Abbildung 5.2 aber zu erkennen ist, lassen sich die Strukturen der `SELECT`-Anweisung zum großen Teil auch zum Aufbau der abstrakten Syntaxgraphen für die anderen Anweisungen nutzen.

Eine `SELECT`-Anweisung muß nun zumindest eine `SELECTION`-Klausel und eine `FROM`-Klausel enthalten. Die Angabe der `WHERE`-, `GROUPBY`- und `HAVING`-Klausel ist optional. Die Knoten der benutzten Klauseln werden über `lc`-Kanten (list contains) mit dem Knoten der jeweiligen Anweisung verbunden. Eine `SELECTION`-Klausel (ebenso wie die `GROUPBY`-Klausel) besteht aus der Aneinanderreihung von Spaltennamen, die durch die Knotenklasse `COLUMN` repräsentiert werden. Diese Knoten werden ebenfalls über `lc`-Kanten mit dem Knoten der zugehörigen Klausel verbunden. Die Reihenfolge, in der die Spalten in der `SELECT`-Anweisung aufgelistet werden, spiegelt sich in `lf` (list first)-, `ln` (list next)- und `ll` (list last)-Kanten wider. Alle oben genutzten Aggregationsbeziehungen geben durch diese Nutzung der `lc`-, `lf`-, `ln`- und `ll`-Kanten sowohl Auskunft über den Struktur einer Anweisung wie auch über die genutzte syntaktische Reihenfolge der jeweiligen Elemente.

Die FROM-Klausel einer Anweisung listet die genutzten Relationen auf, die gegebenenfalls mit Hilfe einer RANGE-Angabe eindeutig identifiziert werden. Da die RANGE-Angabe optional ist und ein COLUMN-Knoten der SELECTION-Klausel eindeutig zugeordnet werden soll, werden RANGE und RELATION generalisiert zu CURSOR. Mit Hilfe einer belongs_to-Kante wird diese eindeutige Zuordnung realisiert. Falls eine RANGE-Angabe in einer Anweisung angegeben ist, wird eine belongs_to-Kante zwischen dem COLUMN-Knoten der SELECTION-Klausel und dem RANGE-Knoten der FROM-Klausel gezogen. Ist keine RANGE-Angabe vorhanden, wird die Kante zwischen dem COLUMN-Knoten und dem RELATION-Knoten gezogen. Diese Kanten gehen über die Darstellung der Syntax hinaus, ermöglichen während der Cliché-Erkennung aber die einfache Zuordnung von Spalten zu Relationen.

Die WHERE-Klausel und die HAVING-Klausel enthalten boolesche Ausdrücke, die als Baum dargestellt werden. Die Struktur des jeweiligen Baums wird wiederum mit Hilfe der lc-, lf-, ln- und ll-Kanten dargestellt (vgl. Abbildung 5.4). Die Spezialisierung der Knotenklasse TREENODE zeigt die folgende Abbildung:

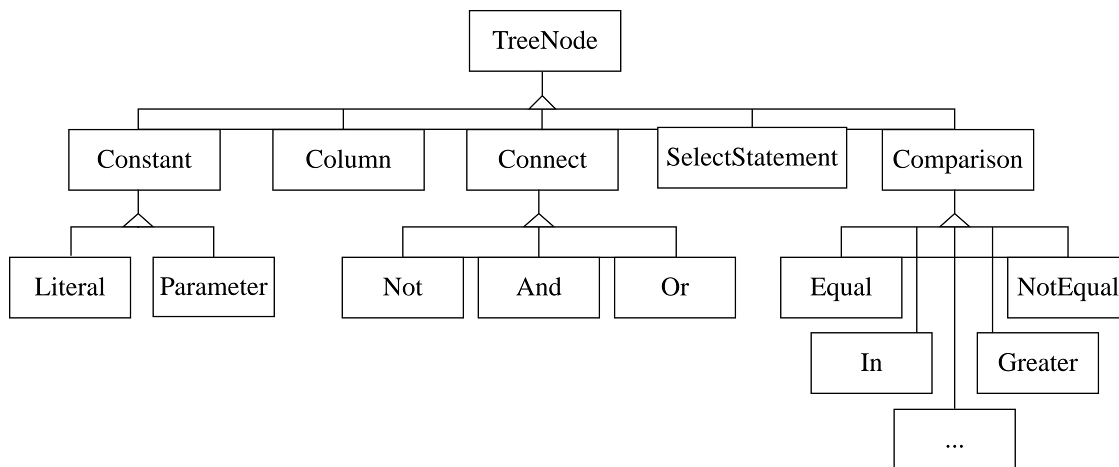


Abbildung 5.3 *Spezialisierung der Knotenklasse TREENODE*

Ein Baum, der die Struktur einer WHERE-Klausel oder HAVING-Klausel widerspiegelt, besteht aus Knoten, die die Verbindung von booleschen Ausdrücken darstellen (CONNECT), Knoten, die Vergleichsoperatoren repräsentieren (COMPARSION) und Knoten für Literale, Parameter und Spalten (COLUMN), über die die Vergleiche durchgeführt werden. Knoten der Klasse COLUMN werden mit Hilfe der oben erwähnten belongs_to-Kante eindeutig einem CURSOR zugeordnet. In einer WHERE-Klausel oder HAVING-Klausel kann aber auch eine vollständige SELECT-Anweisung als sogenannte *Subquery* enthalten sein. Der SelectStatement-Knoten kann also auch ein Teil des Baums sein.

Einen abstrakten Syntaxgraphen für die folgende Beispielanweisung zeigt Abbildung 5.4.

```
SELECT V.Name
   FROM Vorlesung V, Professor P
   WHERE P.Name = 'Schäfer'
```

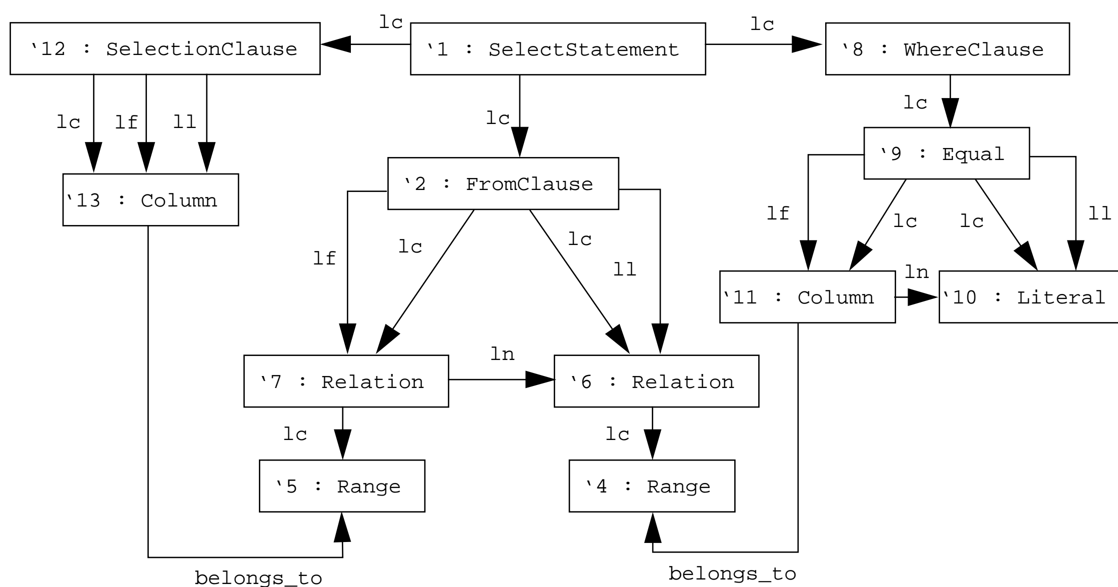


Abbildung 5.4 *Abstrakter Syntaxgraph für eine Beispielanweisung*

Die Clichébibliothek

In der Clichébibliothek werden alle Clichés mit Hilfe von Graphgrammatiken spezifiziert. So ist eine formale Grundlage gegeben, die eine Spezifikation der Clichés auf einem hohen Abstraktionsniveau erlaubt. Die grafische Beschreibung ermöglicht zudem eine Abstraktion von syntaktischen Variationen (vgl. Kapitel 4.2). Zur Beschreibung der Clichés werden geschichtete Graphgrammatiken eingesetzt, die von Rekers und Schürr in [RS97] und [RS95] entwickelt wurden und in Kapitel 5.2 vorgestellt werden. Diese Klasse von Graphgrammatiken ist mächtiger als kontextfreie Graphgrammatiken, wie sie zum Beispiel von Linda Wills in [RW90] benutzt werden, und bieten somit einige Vorteile, die in Kapitel 5.2 näher erläutert werden.

An dieser Stelle soll noch keine genaue Beschreibung der Clichés erfolgen. Sie kann Kapitel 6 entnommen werden.

Der Clichéerkenner

Der Clichéerkenner wendet die spezifizierten Graphgrammatiken auf dem abstrakten Syntaxbaum der SQL-Anweisungen an. Dabei wird ein einfacher Analysealgorithmus angewendet, der die Produktionen der Cliché-Grammatiken nacheinander anwendet, solange dies möglich ist. Durch die Spezifikation der Clichés als geschichtete Graphgrammatiken wird eine Terminierung des Algorithmus garantiert [RS95].

Das Format der erkannten Clichés

Die erkannten Clichés werden in einer Textdatei ausgegeben, die von weiteren Analysewerkzeugen eingelesen werden kann. Diese Datei enthält zeilenweise die einzelnen Clichés mit Namen und den semantischen Informationen, die sich aus der Art des Clichés ableiten lassen.

In Kapitel 4.3 wurden die Clichés im Bereich relationaler Datenbanken in drei Klassen eingeteilt. Clichés zur Erkennung intrarelationaler Abhängigkeiten geben Aufschluß über Abhängigkeiten innerhalb einer Tabelle. Clichés dieser Klasse werden folgendermaßen dargestellt:

```
Name des Clichés( Tabelle, [{] Attribut1 [,Attribut2,..}] )
```

Clichés zur Erkennung interrelationaler Abhängigkeiten beschreiben Beziehungen zwischen Attributen verschiedener Tabellen. Die Ausgabe dieser Clichés erfolgt als:

```
Name des Clichés( [{] (Tabelle1.Attribut1, Tabelle2.Attribut1)
                [,(Tabelle1.Attribut2, Tabelle2.Attribut2),..}] )
```

Die hier betrachteten Clichés zur Erkennung von Optimierungsstrukturen schließlich beziehen sich auch jeweils nur auf *eine* Tabelle, so daß sich die Ausgabe analog zu den Clichés zur Erkennung von intrarelationaler Abhängigkeiten ergibt.

5.2 Geschichtete Graphgrammatiken

Zur formalen Spezifikation der Clichés sollen geschichtete Graphgrammatiken [RS97] eingesetzt werden, die dann im Clichéerkenner ausgeführt werden. Es soll deshalb hier ein Überblick über diese Klasse von Graphgrammatiken gegeben werden. Einen Überblick über Graphgrammatiken im allgemeinen geben [NAG79] oder [ROS97].

In einer textuellen Grammatik ist die Ersetzung eines Nichtterminals durch ein entsprechendes Terminalsymbol bzw. ein anderes Nichtterminal wegen der sequentiellen Reihenfolge der einzelnen Sprachelemente klar definiert. Bei graphischen Sprachen ist dies nicht so einfach, da zwischen den Sprachelementen viele Beziehungen möglich sind. Die ersetzten Elemente sind richtig in den Kontext des Wirtsgraphen einzusetzen. Jede Graphgrammatik muß zunächst eine Lösung für eine korrekte Einbettungsüberführung anbieten (*embedding problem*). Bei kontextfreien Graphgrammatikregeln, deren linke Seite nur aus einem nichtterminalen Knoten besteht, ist die Überführung einfach. Allerdings schränken sie die Ausdrucksfähigkeit einer Grammatik erheblich ein. Geschichtete Graphgrammatiken erlauben deshalb kontextsensitive Graphgrammatikregeln, bei denen beide Regelseiten zunächst beliebige Graphen sein können. Es stellt sich damit also die Frage nach einer korrekten Einbettungsüberführung und vor allem auch nach einer Einbettungsüberführung die eine Terminierung der Ausführung der Graphgrammatik sicherstellt.

Die Korrektheit der Einbettungsüberführung wird in geschichteten Graphgrammatiken durch sogenannte *Kontextelemente* sichergestellt. Diese Kontextelement sind Knoten und Kanten die in beiden Regelseiten zu finden sind, so daß sich hieraus die korrekte Einbettung in den Wirtsgraphen ergibt.

Bei der Einbettungsüberführung ergeben sich folgende Probleme:

(1) Eindeutige Zuordnung (*identification condition*)

Die Ausführung einer Regel darf nicht dazu führen, daß einem Element der linken Seite, das auf der rechten Regelseite auch nur einmal genutzt wird, während der Ausführung der Produktion mehrere Elemente zugeordnet werden.

(2) Nicht zuzuordnende Kante (*dangling edge condition*)

Es ist möglich, daß während der Ausführung einer Produktion Kanten ungültig sind, die ein Element des Wirtsgraphen mit einem Element, das in der Regel genutzt wird, verbinden sollen, da die Kanten diesen bestimmten Elementtyp nicht als Quell- bzw. Zielelement kennen oder das Element nicht mehr existiert.

Durch das Verhindern der Ausführung dieser Produktionen, die nicht eindeutige bzw. falsche Ergebnisse liefern, werden diese Probleme umgangen. Mit der Berücksichtigung dieser Bedingungen ist eine geschichtete Graphgrammatik auch umkehrbar *(reversible)* [RS97].

Um dem Anwender von geschichteten Graphgrammatiken eine möglichst einfache Spezifikation eigener Regeln zu ermöglichen, unterstützen geschichtete Graphgramma-

tiken die Angabe von Attributen für Knoten und Kanten und sog. *label wildcards*[1]. Ein Bezeichner `compare` z.B. könnte in Graphgrammatiken für den oben definierten Syntaxgraphen ein Element aus der Menge aller Knoten repräsentieren, die Vergleichsoperatoren darstellen: `compare ∈ {GREATER, LESS, EQUAL, NOTEQUAL,...}`. Dabei soll hier zur Vereinfachung (analog zu [RS97]) eine Produktion, in der Wildcards genutzt werden, für die entsprechende Anzahl von Produktionen stehen, in der die Wildcards durch entsprechende Elemente ersetzt werden.

Herkömmlichen kontextsensitiven Graphgrammatiken bereitet es Probleme, die Terminierung der Ausführung der Grammatik sicherzustellen. In geschichteten Graphgrammatiken muß deshalb die linke Seite einer Produktion lexikographisch kleiner als die rechte Seite sein, so daß keine zyklischen Abhängigkeiten entstehen. Dabei wird eine lexikographische Ordnung durch die Definition sogenannter Schichten *(Layer)* hergestellt, d.h. die Bezeichner werden nicht einfach eingeteilt in Terminale und Nichtterminale. Statt dessen wird die Menge aller Knoten- und Kantenbezeichner $L_V \otimes L_E$[2] aufgeteilt in n disjunkte Mengen $L_0 \otimes \ldots \otimes L_n$ mit einer zugehörigen Funktion `layer`, die für alle Elemente eines Graphen den Index der Schicht angibt, zu der das Element gehört.

Eine Graphgrammatik nennt sich dann geschichtete Graphgrammatik in Bezug auf die Aufteilung $L_0,\ldots, L_n$ aller Bezeichner, wenn

- die rechte Seite R einer Produktion ein verbundener Graph,
- die linke Seite L einer Produktion nicht leer,
- die rechte Seite R ohne die Elemente von L nicht leer ist und
- L < R ist, wobei folgende lexikographische Ordnung der Graphen gilt:

  $$G < G' :\Leftrightarrow \exists i : |G|_i < |G'|_i \wedge \forall (j < i) : |G|_j = |G'|_j$$

  wobei $|G|_k$ definiert ist als $|\{ x \in G \mid layer(x) = k\}|$, d.h. $|G|_k$ ist die Anzahl der Elemente in G, die der Schicht L_k angehören.

Mit dieser Definition terminiert jeder einfache Parsingalgorithmus, der die linken und rechten Seiten der Produktionen so lange austauscht, wie es möglich ist und falls notwendig, fehlgeschlagene Produktionen zurücksetzt (vgl. [RS97], [RS95]).

1. Mehrere Knoten- bzw. Kantentypen werden mit Hilfe von *label wildcards* unter einem Bezeichner zusammengefaßt. Bei Angabe dieses Bezeichners in der Grammatik wird zur Ausführung ein beliebiger Knoten- bzw. Kantentyp aus der zusammengefaßten Menge angesprochen.

2. $x \otimes y$ ist die disjunkte Vereinigung von Mengen

„A naive parsing algorithm, which applies productions with exchanged left- and right-hand sides as long as possible and backtracks when necessary, terminates always and produces the correct answer“ [RS97].

5.3 Realisierung des Clichéparsers

Im Rahmen dieser Arbeit entstand der Prototyp eines Clichéparsers für relationale Datenbankanwendungen, der zur Cliché-Erkennung in der Datenbank Reengineering Umgebung VARLET eingesetzt werden soll.

Das zu untersuchende Programm, in dem die Datenbankzugriffe in Embedded-SQL formuliert sind, sollte als Textdatei vorliegen. Aus dieser Datei werden dann mit Hilfe eines SQL-Parsers die Embedded-SQL-Zugriffe herausgefiltert. Dieser Parser ist realisiert mit den UNIX-Werkzeugen *Lex* und *Yacc*, die eine lexikalische (Lex) und syntaktische Analyse (Yacc) der Eingabedatei vornehmen. Die lexikalische Analyse zerlegt die Datei, „indem sie bestimmte Textteile nach vorgegebenen Regeln erkennt und klassifiziert“ [HER92]. Die lexikalische Analyse des realisierten Parsers erkennt dabei lediglich die Embedded-SQL-Anweisungen. Der restliche Programmcode wird überlesen. Die erkannten Textteile werden einer Syntaxanalyse unterzogen. Diese wird mit dem Werkzeug Yacc realisiert, das eine Beschreibung der Syntax in Backus-Naur-Form verlangt. Ein Ausschnitt aus der Syntaxbeschreibung der SQL-Anweisungen ist dem Anhang A zu entnehmen.

Korrekt erkannte Strukturen werden zunächst zwischengespeichert, bis die lexikalische und syntaktische Analyse abgeschlossen ist und somit sichergestellt ist, daß alle in der Eingabedatei gefundenen Embedded-SQL-Anweisungen eine korrekte Struktur haben. Die Strukturen, die zur Zwischenspeicherung genutzt werden, orientieren sich bereits an denen, die im abstrakten Syntaxgraphen verwendet werden. Dabei werden z.B. die `RELATION`- und `RANGE`-Angaben, die bei der syntaktischen Analyse einer `FROM`-Klausel eines `SELECT`-Statements zugeordnet werden, in einer Liste aller Anweisungen gespeichert, diese enthält neben einem Hinweis auf die Art der Anweisung (`update`, `delete`, `select`) einen Zeiger auf die Struktur `stmt_parts`, aus denen die Anweisung bestehen kann.

```
struct stmt_parts{
  boolean           all;
  boolean           distinct;
  column_list      *selection;
  relation_list    *from;
  operation_list   *where;
  column_list      *group_by;
  operation_list   *having;
}

mit:

struct relation_list {
  char          relation[MAXSTRING];      // Name der Relation
  char          range[MAXSTRING];         // Bezeichnung des Range
  relation_list *next;
};

struct column_list {
  char          column[MAXSTRING];        // Name der Spalte
  column_list   *next;
};

struct operation_list {
  char             operation[MAXSTRING];  // genaue Bezeichnung der
                                          // Operation z.B.'=' für
                                          // Knotentyp COMPARISON_NODE
  ProgresNodeTypes  nodetype;
  operation_list   *next;
};

typedef enum {
  COLUMN_NODE,                 // Attribute einer Relation
  LITERAL_NODE,                // Literal
  PARAMETER_NODE,              // Parameter
  BETWEEN_NODE,
  LIKE_NODE,
  AND_NODE,
  OR_NODE,
  NOT_NODE,
  IN_NODE,
  COMPARISON_NODE,             // EQUAL, GREATER, LESS,...
  SUBQUERY_NODE                // Anker für Subqueries
}ProgresNodeTypes;
```

Dabei sind die einzelnen Knoten einer `operation_list` in UPN (umgekehrt polnischer Notation) verkettet, aus der sich die korrekte Struktur des Baums ergibt.

Zum Aufbau des Graphen wird die graphische Spezifikationssprache und Softwareentwicklungsumgebung Progres (PROgrammierte GRaphErsetzungsSysteme) genutzt. Die Implementierung der Clichés erfolgt ebenfalls in Progres.

Einen Überblick über den Clichéerkenner gibt folgende Abbildung:

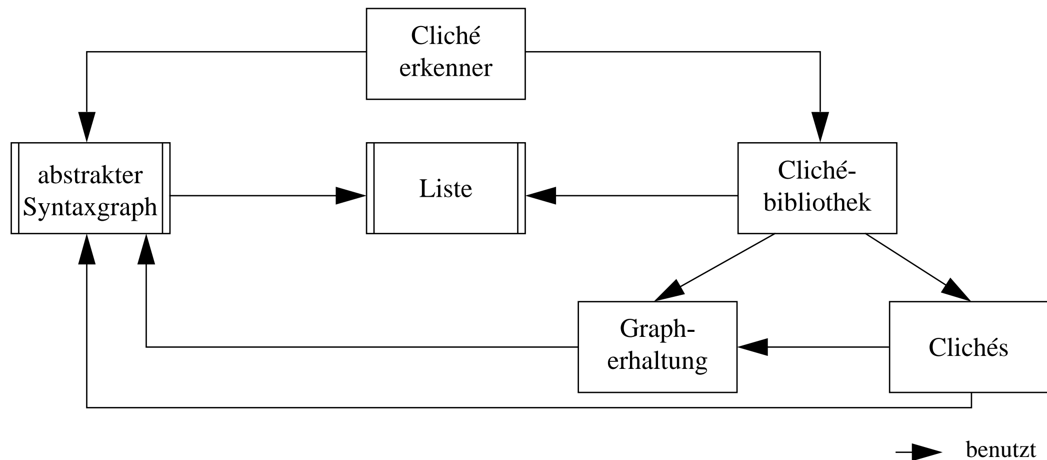


Abbildung 5.5 *Architektur des Clichéerkenners*

Der `Clichéerkenner` nutzt den vom Clichéparser erstellten abstrakten Syntaxgraphen der DML-Anweisungen und übernimmt die Ausführung der implementierten Produktionen. Die entsprechende Funktion ist im Modul Clichébibliothek implementiert, die einen einfachen Algorithmus enthält, der, so lange dies möglich ist, nacheinander alle Produktionen ausführt. Als Bedingung für die korrekte Terminierung eines Parsingalgorithmus für geschichtete Graphgrammatiken, die im Modul `Clichés` implementiert sind, wurde in Kapitel 5.2 das Backtracking bei fehlgeschlagenen Produktionen genannt. Dies ist eine Funktionalität, die von Progres standardmäßig genutzt wird, so daß die korrekte Terminierung des Clichéerkenners gegeben ist.

Das Modul `Clichébibliothek` stellt Funktionen zur Verfügung, die die Ausführung der im Modul `Clichés` spezifizierten Graphgrammatiken durchführen. Die in Kapitel 6 gegebenen Clichéspezifikationen beschränken sich auf das Erkennen der jeweiligen Clichés. Eine identische Implementierung würde dazu führen, daß mit jeder Ausführung einer Regel und der entsprechenden Graphersetzung, ein Teil der Struktur der abstrakten Syntaxgraphen verloren ginge. Da alle Syntaxgraphen aber nach mehreren Clichés durchsucht werden sollen, ist durch entsprechende „grapherhaltende Erweiterungen“ bei der Implementierung sicherzustellen, daß nach der Ausführung einer Regel, der Syntaxgraph in seiner Struktur unverändert bleibt. Aus diesem Grund stellt das Modul `Grapherhaltung` Funktionen bereit, die sicherstellen, daß der Ausführungsalgorithmus des Moduls `Clichéerkenner` terminiert, obwohl der Ausgangsgraph unverändert bleibt. Dieser Mechanismus wird im folgenden Kapitel 5.3.1 näher erläutert.

Der implementierte Clichéparser stellt schließlich weiteren Analysewerkzeugen eine Textdatei mit den erkannten Clichés zur Verfügung. Das Format der Ausgabe ist in Kapitel 5.1 beschrieben.

5.3.1 Unterschiede zwischen Konzeption und Realisierung

Dieses Kapitel geht auf die Unterschiede zwischen der Konzeption der Clichéspezifikation und deren Implementierung ein.

Das Modul `Grapherhaltung` (vgl. Abbildung 5.5) soll eine Terminierung der Ausführung der Graphgrammatiken sicherstellen und dafür sorgen, daß der abstrakte Syntaxgraph unverändert bleibt. Die Funktionsweise soll hier am Beispiel der Produktion 1 zur Erkennung des Select-Distinct Clichés (vgl. Kapitel 6.1.1) genauer beschreiben werden.

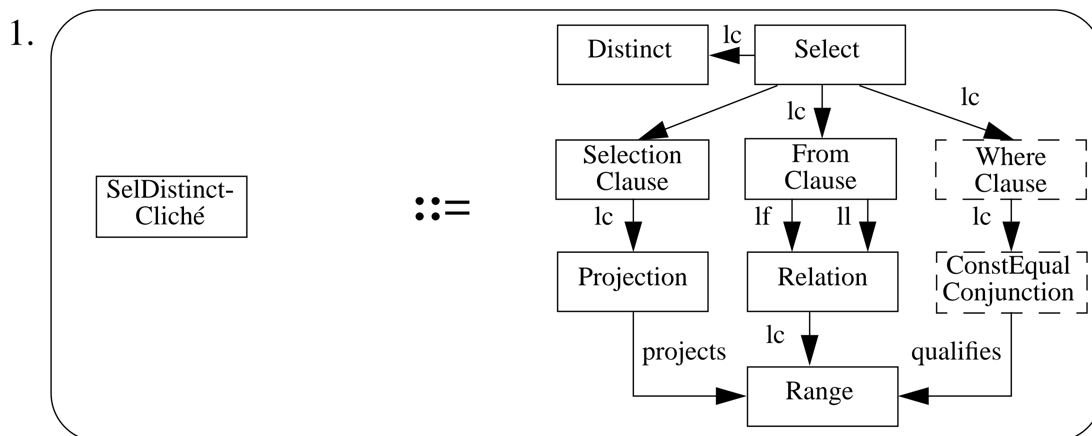


Abbildung 5.6 *Spezifikation einer Produktion*

Diese Produktion nutzt die aus Progres bekannten *optionalen Knoten* (dargestellt durch gestrichelte Knoten), die in geschichteten Graphgrammatiken nicht verwendet werden. Dies soll die Spezifikation vereinfachen. Dabei steht eine Produktion, die optionale Knoten enthält, für zwei Produktionen, bei denen die eine diese Knoten enthält und die andere nicht. Bei einer exakten Implementierung dieser Produktion würde die rechte Regelseite durch die linke ersetzt, so daß der Graph der rechten Regelseite verlorengeht. Da die Cliché-Erkennung aber nach verschiedenen Clichés sucht, ist es wichtig, daß der Syntaxgraph unverändert bleibt. Aus diesem Grund übernimmt bei der Implementierung die linke Regelseite alle Elemente der rechten Seite und paßt die linke Regelseite der spezifizierten Produktion entsprechend in den Graph ein. Für die oben beschriebene

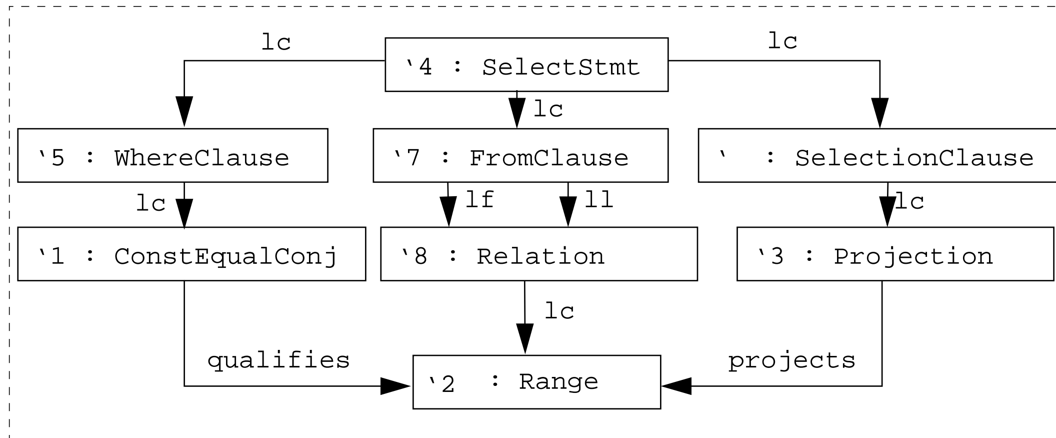
Produktion ergibt sich die Progres-Implementierung mit Hilfe der Regeln auf den folgenden Seiten. Da in Progres optionale Knoten nicht mit Restriktionen versehen werden dürfen, muß die oben gegebene Regel mit zwei Progres-Produktionen implementiert werden. Die erste Produktion enthält die optionalen Knoten, die zweite nicht. Wie man aber erkennt, wird der obere Graph komplett in die linke Regelseite (unterer Graph der Produktion) übernommen. Diese Seite wird dann noch um den Knoten zur Erkennung eines Select-Distinct Clichés erweitert. Diese Vorgehensweise wurde bei allen Produktionen realisiert, so daß die Ausführung einer Produktion den Syntaxgraphen nicht verändert.

Das Modul `Clichéerkenner` arbeitet für diese Produktion folgendermaßen:

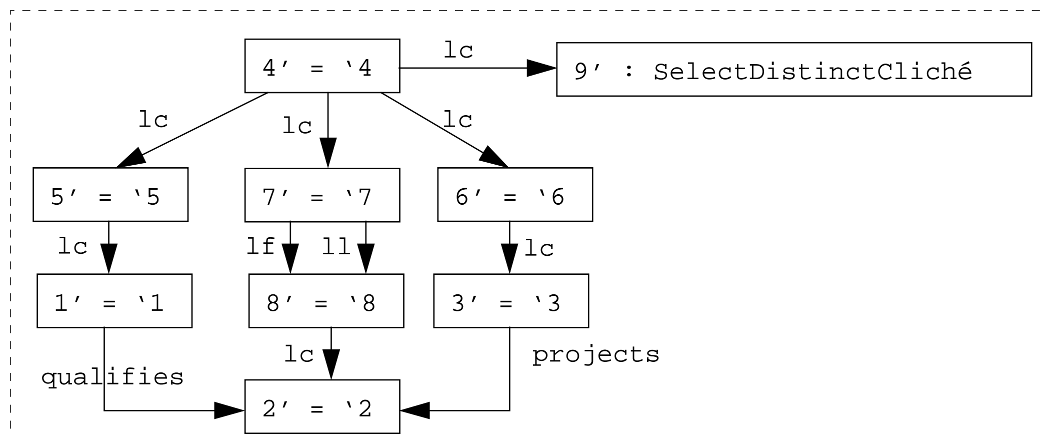
```
loop
     choose
       DistinctGrammar_Prod1
       ( out replacedNodes, out replacedbyNodes )
     else
       DistinctGrammar_Prod11
       ( out replacedNodes, out replacedbyNodes )
     end
   & MarkReplacedNodes
     ( replacedNodes, replacedbyNodes, "Distinct-Production1" )
 end
```

Der abstrakte Syntaxgraph wird nach allen Strukturen durchsucht, die der rechten Regelseite entsprechen. Dabei werden zunächst alle Strukturen ersetzt, die die optionalen Knoten enthalten. Für alle erfolgeichen Ausführungen der beiden Produktionen, wird mit Hilfe der Funktion `MarkReplacedNodes` des Moduls `Grapherhaltung` markiert, welche Struktur durch welchen Knoten ersetzt wurde.

```
production DistinctGrammar_Prod1
          ( out replacedNodes : ReplaceableNode [1:n] ;
            out replacedbyNodes : TreeNode [1:n])
        [0:n] =
```

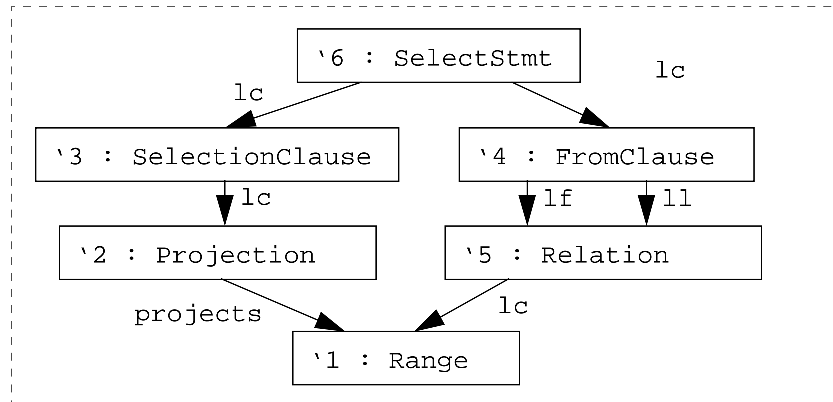



```
        ::=
```

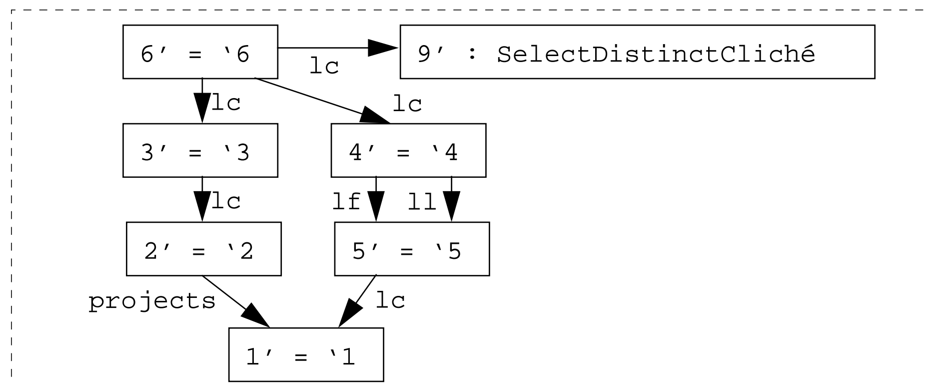



```
        condition '4.Distinct = true;
          empty ( '1.CheckIfReplaced ( "Distinct-Production1" ) );
          empty ( '2.CheckIfReplaced ( "Distinct-Production1" ) );
          empty ( '3.CheckIfReplaced ( "Distinct-Production1" ) );
          empty ( '8.CheckIfReplaced ( "Distinct-Production1" ) );
        transfer
          9'.ClichesAttributes := '1.Attributes & ", " & '3.Attributes;
          9'.NoOfAttr := '1.cardAttr + '3.cardAttr;
          9'.ClichesTable := '3.Relation;
        return replacedNodes := '1 or '2 or '3 or '8;
               replacedbyNodes := 9';
      end;
```

```
production DistinctGrammar_Prod11
           ( out replacedNodes : ReplaceableNode [1:n] ;
             out replacedbyNodes : TreeNode [1:n])
        [0:n] =
```




```
        ::=
```

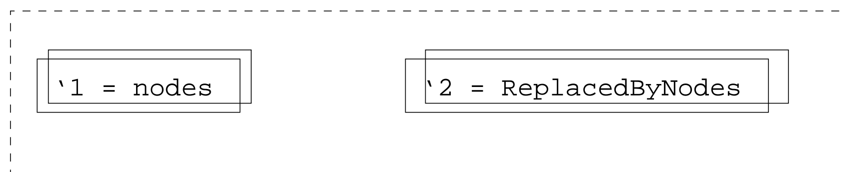



```
        condition `6.Distinct = true;
        empty ( `1.CheckIfReplaced ( "Distinct-Production1" ) );
        empty ( `2.CheckIfReplaced ( "Distinct-Production1" ) );
        empty ( `5.CheckIfReplaced ( "Distinct-Production1" ) );
        transfer
           9’.ClichesAttributes := `2.Attributes;
           9’.NoOfAttr := `2.cardAttr;
           9’.ClichesTable := `2.Relation;
        return replacedNodes := `1 or `2 or `5;
               replacedbyNodes := 9’;
      end;
    end;
```

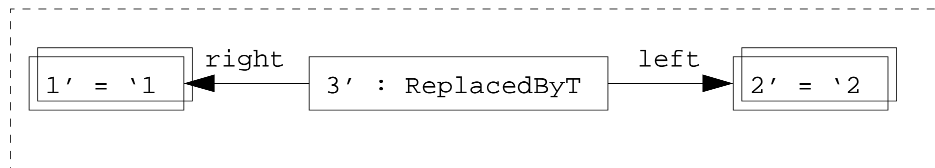
Die Erhaltung des Syntaxgraphen in seiner ursprünglichen Struktur bringt nun aber das Problem mit sich, daß „manuell“ sichergestellt werden muß, welche Teile eines Graphen bereits ersetzt wurden, um eine Terminierung zu gewährleisten. Das Modul `Graphererhaltung` stellt deshalb Funktionen zur Verfügung, die es ermöglichen, Knoten als ersetzt zu markieren bzw. zu prüfen, ob ein Knoten mit einer bestimmten Produktion noch ersetzt werden darf.

Die Markierung einer vorgenommen Ersetzung erfolgt nach erfolgreicher Ausführung einer Produktion mit Hilfe der Funktion `MarkReplacedNodes`. Diese Funktion verlangt als Eingabeparameter die Knoten, die ersetzt wurden, die Knoten durch die ersetzt wurde und den Namen der Produktion. Die Knoten, die ersetzt wurden sollten von der Klasse `ReplaceableNode` abgeleitet sein. Zur Zeit gilt dies für alle Knotenklassen. Die Markierung erfolgt dann mit Hilfe eines Knotens vom Typ `ReplacedBy`, der auf die Knoten entsprechend verweist und zur eindeutigen Identifizierung ein Attribut mit dem Namen der ersetzenden Produktion erhält.

```
production MarkReplacedNodes( nodes : ReplaceableNode [1:n] ;
                              ReplacedByNodes : TreeNode [1:n];
                              Production : string)
    =
```




```
      ::=
```




```
      transfer 3'.ReplaceProduction := Production;
    end;
```

Die Restriktion `CheckIfReplaced` prüft, ob ein gegebener Knoten bereits durch die gegebenen Produktion ersetzt wurde. Eine bereits erfolgte Ersetzung eines Knotens durch die gegebene Produktion wird festgestellt, wenn bereits eine eingehende <-right-Kante existiert, die auf einen `ReplacedBy`-Knoten verweist, dessen Attribut mit der gegebenen Produktion übereinstimmt. Falls dies für einen zu prüfenden Knoten der Fall ist, wird die Ausführung dieser Produktion abgebrochen. Dabei ist zu beachten, daß alle Knoten einer zu ersetzenden Struktur mit dieser Restriktion geprüft werden.

Da sich die Spezifikationen in Kapitel 6 auf das Erkennen der Clichés beschränken, ohne dabei die beteiligten Tabellen und Attribute zu berücksichtigen, erfordert die Implementierung eine Erweiterung in dieser Richtung. Wie im Beispiel zu sehen, werden deshalb bei der Implementierung der einzelnen Clichéklassen entsprechende Attribute definiert, die bei der Ausführung der Produktion mit den zugehörigen Werten gefüllt werden. Die Möglichkeit in Progres, Knoten durch Attribute genauer zu beschreiben, kann die Implementierung außerdem vereinfachen. So zeigt das oben beschriebene Beispiel, daß ein

Knoten `Distinct` als Attribut eines `SelectStatement`-Knotens realisiert werden kann.

Zwischen den Spezifikationen der Clichés als geschichtete Graphgrammatiken und der Implementierung in Progres gibt es aber auch Abweichungen, die aus den unterschiedlichen Funktionalitäten resultieren, die die Spezifikation in Progres bzw. geschichteten Graphgrammatiken bieten. In geschichteten Graphgrammatiken z.B. wird mit Label Wildcards gearbeitet, die in Progres in dieser Form nicht bekannt sind. Die Wildcards wurden deshalb als Superklasse der Klassen implementiert, die durch die jeweiligen Wildcards repräsentiert werden können. Die in Kapitel 6.1.1 definierte Wildcard-Knotenklasse `DefiniteConstant ∈ {Literal, Parameter}` wurde demnach als Superklasse der Knotenklassen `Literal` und `Parameter` implementiert.

5.3.2 Erweiterung der Clichébibliothek

Eine Erweiterung der Clichébibliothek erfordert die Implementierung der spezifizierten Regeln als Progres-Produktionen im Modul `Clichés`. Dabei sind die oben beschriebenen grapherhaltenden Erweiterungen der Produktionen zu berücksichtigen. Zudem ist für jeden Knoten, der in einer Regel ersetzt werden soll, mit Hilfe der Funktion `CheckIfReplaced` zu prüfen, ob eine Ersetzung möglich ist. Zudem sollte jede Produktion folgende Ausgabeparameter enthalten:

```
out replacedNodes : ReplaceableNode [1:n] ;
out replacedbyNodes : TreeNode [1:n]
```

Dabei bezeichnet `replacedNodes` die Menge der Knoten, die in dieser Produktion ersetzt wurden und `replacedbyNodes` die Menge der Knoten, durch die die ersten ersetzt wurden. Dabei sollten die `replacedNodes`-Knoten von der Klasse `ReplaceableNode` erben.

Der Algorithmus im Modul `Clichébibliothek` ist dann um eine Funktion zu erweitern, die die Aufrufe der neu implementierten Produktionen enthält. Nach erfolgreicher Ausführung einer Produktion ist die Funktion `MarkReplacedNodes` mit `replacedNodes` und `replacedbyNodes` aus dem Modul `Grapherhaltung` aufzurufen. Die so implementierte Funktion zur Erkennung des neuen Clichés ist dann in der Funktion `DetectAllCliches` des Moduls Clichébibliothek aufzurufen, da standardmäßig nur die `DetectAllCliches`-Funktion vom Modul `Clichéerkenner` aufgerufen wird.

6 Spezifikation der Clichés

In diesem Kapitel erfolgt eine formale Spezifikation typischer Clichés aus dem Bereich relationaler Datenbankanwendungen, um semantische Informationen aus den Anfragen an die Datenbank zu gewinnen. Dabei wird von einem abstrakten Syntaxgraphen der SQL-Anweisungen ausgegangen, wie er in Kapitel 5.1 beschrieben ist. Die Spezifikation erfolgt mit Hilfe von Layered Graphgrammatiken, die in Kapitel 5.2 vorgestellt wurden. Es werden hier zur Vereinfachung der Darstellung zusätzlich die aus Progres-Spezifikationen bekannten *optionalen Knoten* (dargestellt durch gestrichelte Knoten) und *Pfadausdrücke* (dargestellt durch breitere weiße Pfeile) benutzt. Dabei soll eine Produktion, die optionale Knoten enthält, für zwei Produktionen stehen, bei denen die eine (alle) diese Knoten enthält und die andere nicht. Ein Parsingalgorithmus versucht dann zunächst, die Produktion mit den optionalen Knoten auszuführen und nur, falls dies nicht möglich ist, wird versucht, die Produktion ohne die optionalen Knoten auszuführen. Pfadausdrücke ersetzten eine bestimmte Menge von Kanten und Knoten. Die in den Spezifikationen benutzten Attribute (dargestellt durch Ziffern rechts neben den Knoten) dienen der eindeutigen Zuordnung von Knoten der rechten Seite zu Knoten der linken Seite. Die Endungen der Knotennamen sollen einen Hinweis auf die Strukturen geben, die der jeweilige Knotentyp repräsentiert. Die Endung `Conjunction` weist auf eine Und-Verknüpfung hin, während `Connect` auf eine allgemeine logische Verknüpfung verweist. Bei der Endung `List` handelt es sich um eine sonstige Zusammenfassung von Knoten. Knotenbezeichner, die mit „`*`“ beginnen, weisen auf ein Label Wildcard hin.

6.1 Clichés zur Erkennung von intrarelationalen Abhängigkeiten

6.1.1 Select-Distinct Cliché

Die Verwendung des Schlüsselwortes `distinct` in einer `SELECT`-Anweisung weist darauf hin, daß mehrere gleiche Tupel zurückgegeben werden können, die in der Ergebnisrelation nicht mehrfach aufgeführt werden sollen.

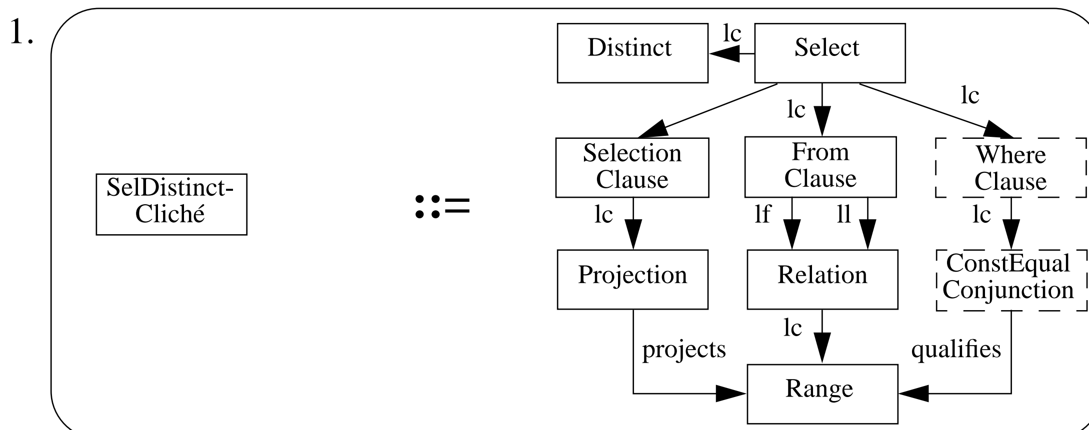
select **distinct** Fachbereich
from Professor
where **Name = :N**

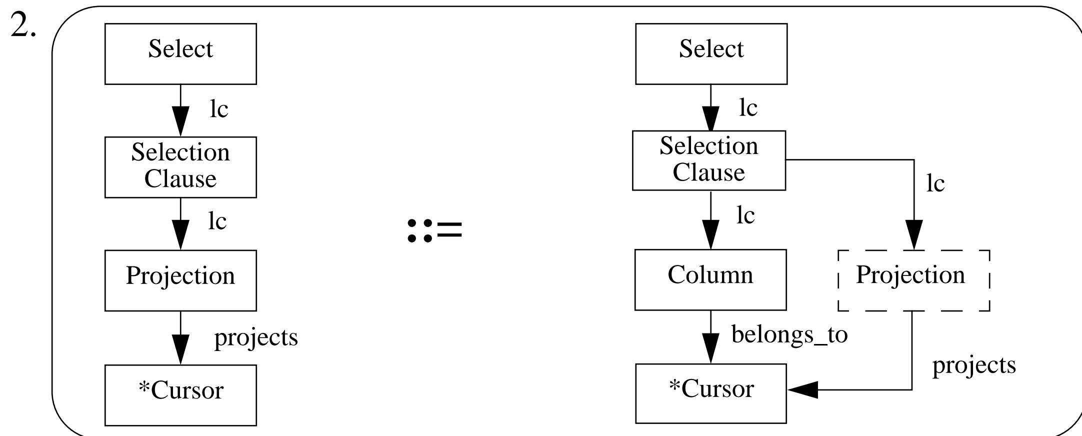
Mit einer Anfrage dieser Art wird der Fachbereich eines Professors gesucht, der einen bestimmten Namen hat. Die `distinct`-Angabe weist zum einen darauf hin, daß die Ergebnisrelation, die in diesem Fall nur das Attribut `Fachbereich` beinhaltet, aus mehreren gleichen Tupeln bestehen kann, so daß davon auszugehen ist, daß dieses Attribut kein Schlüssel ist. Des weiteren wird das Attribut `Name` mit einem konstanten Wert verglichen. Auch hier weist die `distinct`-Angabe darauf hin, daß der Entwickler keinen eindeutigen Rückgabewert erwartet und das Vergleichsattribut `Name` somit kein Schlüssel ist. Dabei können diese Aussagen nur gemacht werden, da in der `FROM`-Klausel nur eine Relation angesprochen wird. Beim Verbund über mehrere Relationen lassen sich diese Rückschlüsse nicht ziehen. Verallgemeinert kann also gesagt werden, daß die Attribute der Ergebnisrelation sowie alle Attribute, die in der `WHERE`-Klausel mit einem konstanten Wert auf Gleichheit verglichen werden, keinen Schlüssel bilden. Bei `WHERE`-Klauseln, die mehrere Vergleiche beinhalten, müssen alle angegebenen Bedingungen Vergleiche auf Gleichheit bestimmter Attribute mit einer Konstanten sein, die durch `Und` verknüpft sind.

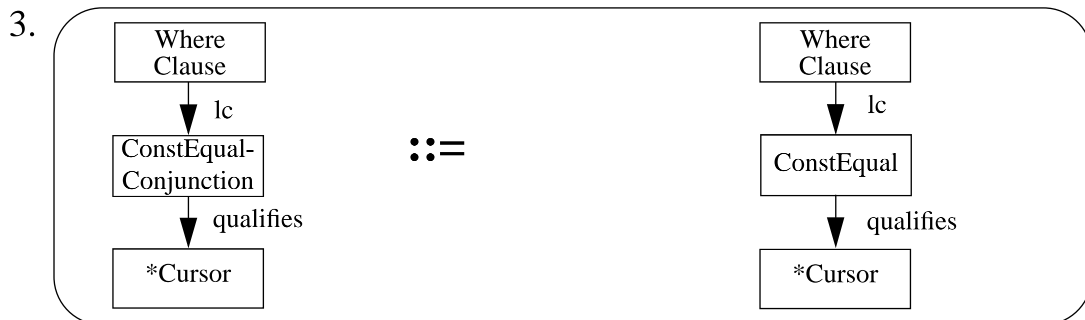
Die Spezifikation nutzt folgende Label Wildcards:

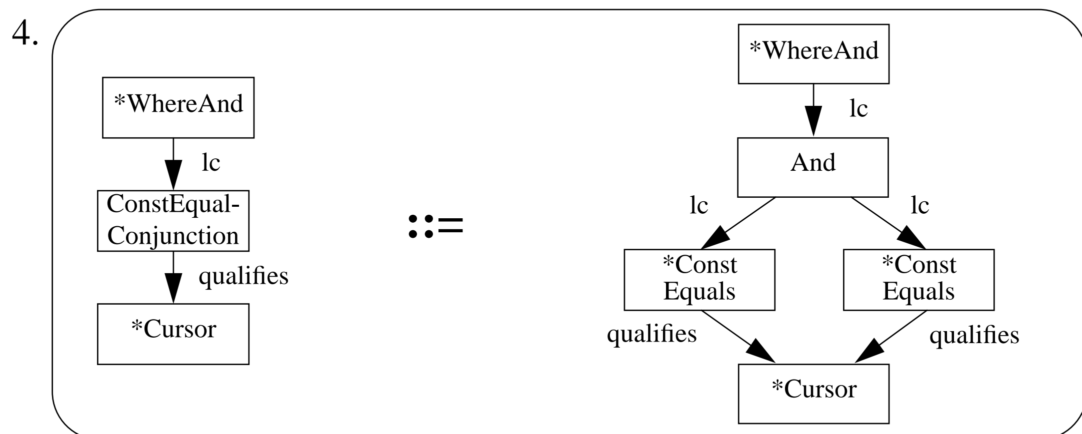
```
*WhereAnd          ∈  {WhereClause, And};
*DefiniteConstant  ∈  {Literal, Parameter};
*ConstEquals       ∈  {ConstEqualConjunction, ConstEqual};
*Cursor            ∈  {Range, Relation};
```

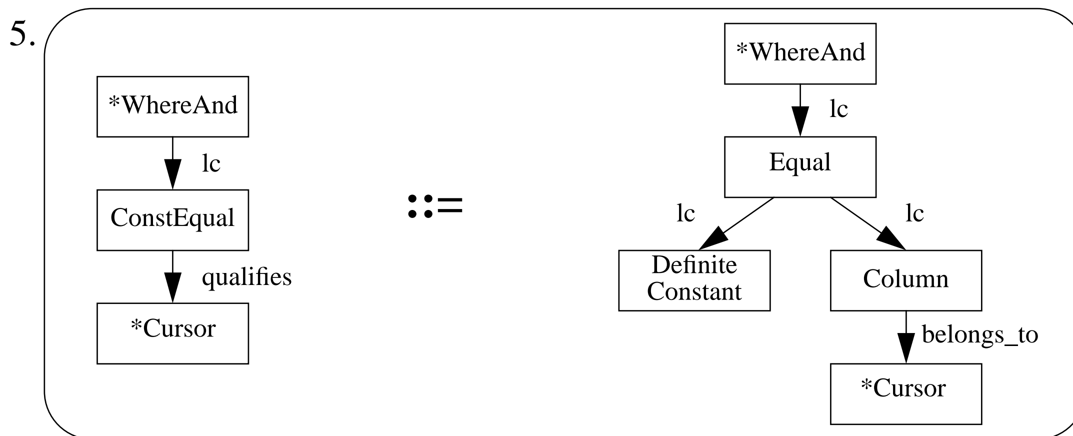
1.




2.
Select
lc
Selection Clause
lc
Projection
projects
*Cursor
::=
Select
lc
Selection Clause
lc
lc
Column
Projection
belongs_to
projects
*Cursor


3.
Where Clause
lc
ConstEqual-Conjunction
qualifies
*Cursor
::=
Where Clause
lc
ConstEqual
qualifies
*Cursor


4.
*WhereAnd
lc
ConstEqual-Conjunction
qualifies
*Cursor
::=
*WhereAnd
lc
And
lc
lc
*Const Equals
*Const Equals
qualifies
qualifies
*Cursor



Die Produktion 5 beschreibt die Suche nach einem Vergleich auf Gleichheit zwischen einem Attribut einer Tabelle und einer Konstanten. Dabei kann in SQL eine Konstante nur in Form eines Literals oder Parameters angegeben werden. Alle passenden Strukturen im abstrakten Syntaxgraphen werden mit Produktion 5 durch den Knotentyp `ConstEqual` ersetzt. Alle `Und`-Verknüpfungen dieser Knoten werden in Regel 4 dann ersetzt durch Konten des Typs `ConstEqualConjunction`. Falls nur ein entsprechender Vergleich gefunden wurde, sorgt die Ausführung der Regel 3 dafür, daß ein Konten des Typs `ConstEqualConjunction` direkt an dem `WhereClause`-Knoten hängt. Nur wenn dies der Fall ist, wurden ausschließlich Gleichheitsbedingungen zwischen einem Attribut und einer Konstanten angegeben. Falls mehrere Vergleiche gefunden wurden, sind diese durch `Und` verknüpft. Die Regel 2 faßt die einzelnen, in der `Selection`-Angabe aufgelisteten Attribute einer `SELECT`-Anweisung auf analoge Art und Weise zusammen. In Regel 1 schließlich wird ein Select-Distinct Cliché erkannt. Da eine `SELECT`-Anweisung auch nur eine `Selection`-Angabe enthalten kann, ist es möglich, daß keine `WHERE`-Klausel gefunden wird. Deshalb sind der `WhereClause`-Knoten und der `ConstEqualConjunction`-Knoten optional. Falls aber eine `WHERE`-Klausel in der `SELECT`-Anweisung angegeben wurde, muß sie die für ein Select-Distinct Cliché notwendigen Eigenschaften aufweisen, d.h. es muß ein `ConstEqualConjunction`-Knoten vorhanden sein. Dabei darf die `FROM`-Klausel allerdings nur eine Relation und, falls vorhanden, auch nur eine `RANGE`-Variable beinhalten. Die oben angegebene Regel 1 beschreibt den Fall, daß die `SELECT`-Anweisung eine `RANGE`-Angabe nutzt.

Um die korrekte lexikographische Ordnung sicherzustellen, werden die Knoten und Kanten der Produktionen[1] folgendermaßen auf zwei Schichten aufgeteilt:

layer(x) = 0: $\forall(x \in \{\{V\} \backslash \{\text{ConstEqualConjunction, Projection}\}\})$

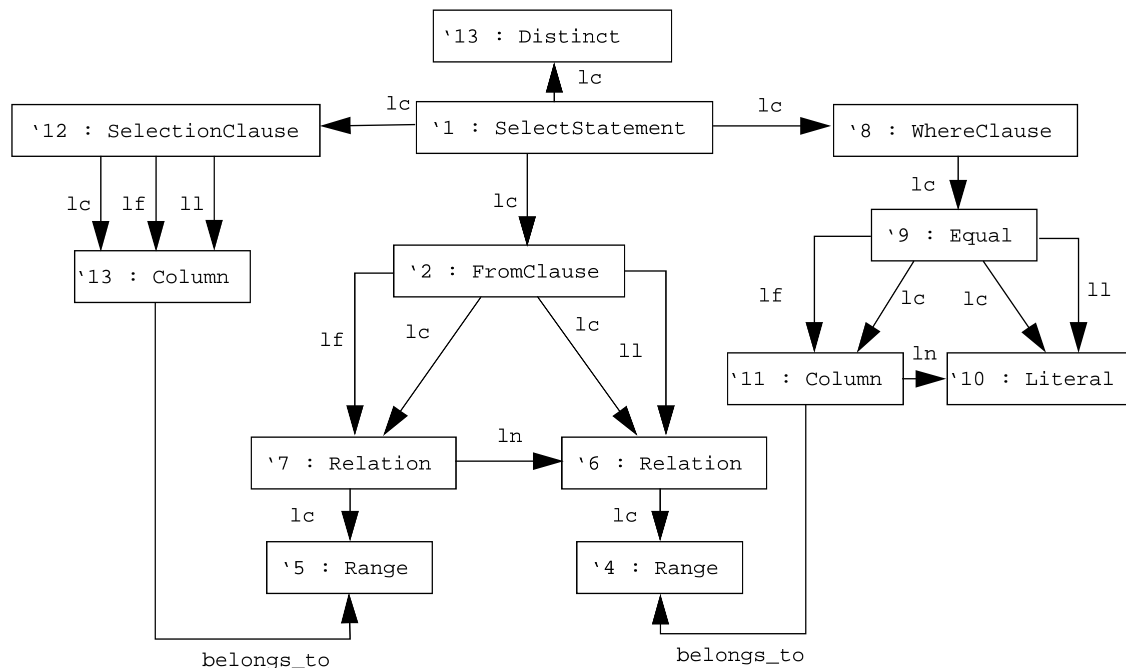
layer(x) = 1: $\forall(x \in \{\{E\}, \text{ConstEqualConjunction, Projection}\})$

Da z.B. Regel 2 optionale Knoten enthält, ist an dieser Stelle eine Überprüfung der lexikographischen Ordnung zum einen für die Regel mit den optionalen Knoten und den zugehörenden Kanten und zum anderen für die Regel ohne die optionalen Knoten und die Ein- und Ausgangskanten durchzuführen. Für diese Regel ergibt sich ohne die optionalen Knoten der ungünstigste Fall mit einem L_0 von 3 und einem R_0 von 4 für die Schicht 0, so daß $L < R$ gilt.

Anhand einer anderen Beispielanfrage, die das Select-Distinct Cliché nicht erfüllt, wird nun exemplarisch beschrieben, wie ein Parsingalgorithmus für die Spezifikation arbeiten sollte:

```
SELECT V.Name
   FROM Vorlesung V, Professor P
   WHERE P.Name = 'Schäfer'
```

Der abstrakte Syntaxgraph dieser Anweisung hat folgende Struktur:



1. Dabei bezeichnet V die Menge aller Knoten und E die Menge aller Kanten der Graphen (vgl.Definition von Graphen in Kapitel 4.2.2).

Zunächst wirde die Regel 5 angewendet, die die Struktur für die Bedingung `P.Name = 'Schäfer'` durch einen Knoten des Typs `ConstEqual` ersetzt. Dieser Knoten ist über eine `lc`-Kante mit dem `WhereClause`-Knoten und über eine `qualifies`-Kante mit dem `Range`-Knoten mit der Nummer 4 verbunden. Regel 4 findet in diesem Beispiel keine Anwendung. Da der `ConstEqual`-Knoten direkt an dem `WhereClause`-Knoten hängt, wird er mit Hilfe der Regel 3 dann durch einen Knoten des Typs `ConstEqualConjunction` ersetzt. Regel 2 ersetzt den `Column`-Knoten der `Selection`-Klausel durch einen `Projection`-Knoten. Weitere Zusammenfassungen finden in diesem Beispiel nicht statt. Die Ausführung der Regel 1 schlägt dann allerdings fehl. Es sind zwar die korrekten Strukturen für die `Where`-Klausel und die `Selection`-Klausel erkannt und auch ein `Distinct`-Knoten ist vorhanden, allerdings verweisen der `Projection`-Knoten und der `ConstEqualConjunction` auf unterschiedliche Relationen.

6.1.2 Cyclic-Exclusion Cliché

Eine typische Art der Anfrage an eine Datenbank beinhaltet die Suche nach Datensätzen mit gleichen Attributwerten. Es kann z.B. vorkommen, daß identische Datensätze in einer Datenbank vorhanden sind, die sich nur durch den Schlüssel unterscheiden, der hier häufig eine automatisch generierte Ordnungsnummer ist. Diese mehrfach vorhandenen Datensätze bezeichnet man auch als Dubletten. Eine Anfrage zur Dublettensuche in der Tabelle `Professor` in der Beispieldatenbank könnte zum Beispeil folgendermaßen formuliert werden:

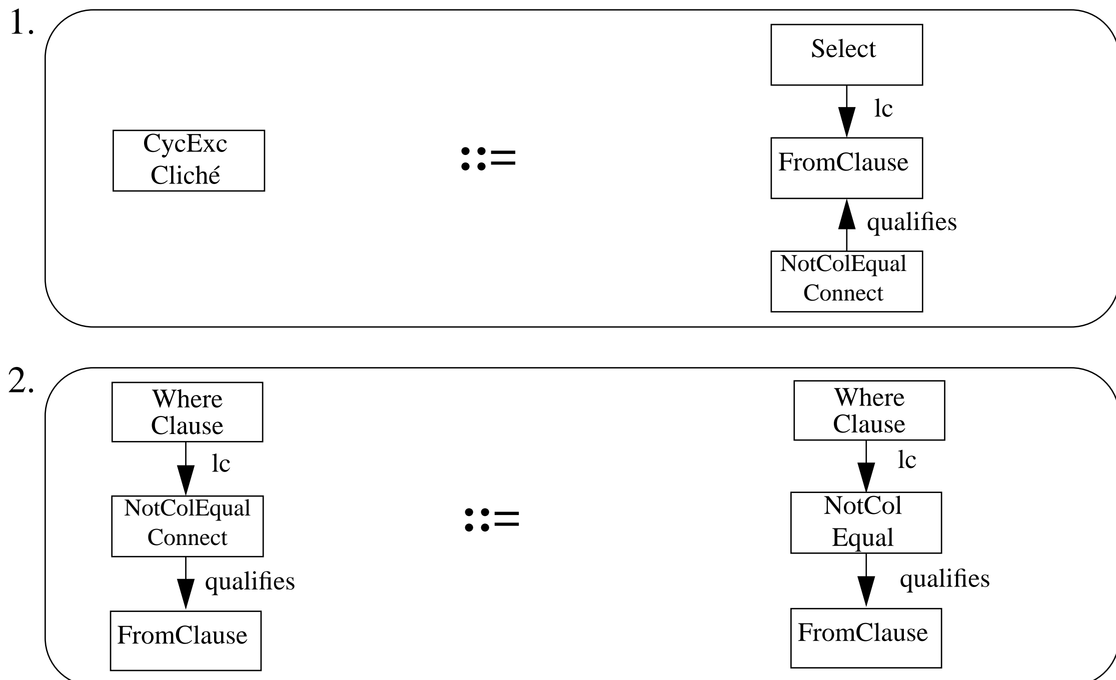
```
select *
  from Professor x, Professor y
  where (x.Name = y.Name)
     and (x.Fachbereich = y.Fachbereich)
     ...
     and not (x.Pers-Nr = y.Pers-Nr)
```

Mit dieser Anfrage werden alle Tupel gesucht, die sich nur durch die Pers-Nr. unterscheiden. Dazu wird ein Verbund über die gleiche Tabelle durchgeführt. Mit der Bedingung `NOT (x.Pers-Nr = y.Pers-Nr)` soll sichergestellt werden, daß der Verbund nicht über vollkommen identische Tupel erfolgt, sondern nur über Tupel, bei denen alle anderen Attributwerte übereinstimmen. Mit dieser Bedingung sollen also Zyklen in der Anfrageabarbeitung verhindert werden. Die eindeutige Identifizierung der Tupel erfolgt über das Attribut `Pers-Nr`, und somit ist dieses Attribut ein Schlüssel-

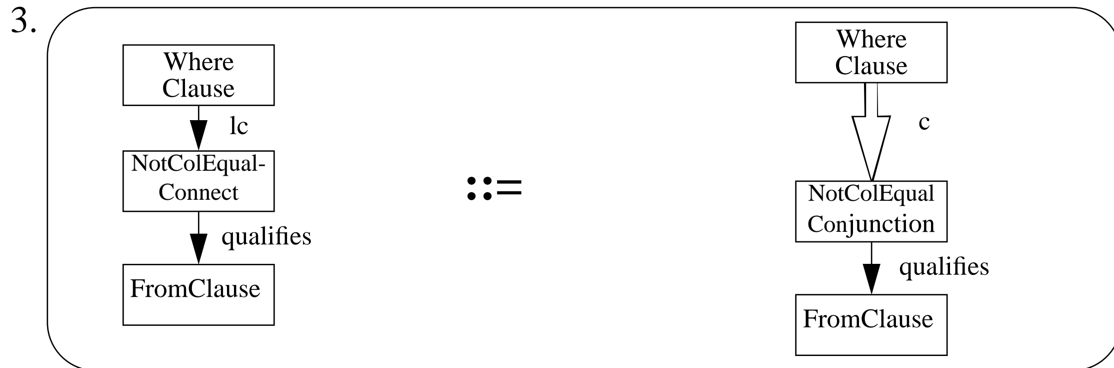
kandidat. Eine Selektion, die paarweise Attribute derselben Tabelle vergleicht und eine Bedingung bzw. mehrere durch eine `Und`-Verknüpfung verbundene Bedingungen der Form `NOT` (`Range1.Attribut` = `Range2.Attribut`) enthält, erfüllt das Cyclic-Exclusion Cliché. Dabei sind `Range1` und `Range2` Rangeangaben, die auf die gleiche Tabelle verweisen (vgl. Regel 5).

In der Regel 3 wird der Pfad `c` benutzt, der definiert ist als `((-lc->)*)`, d.h. die Knoten sind über beliebig viele `lc`-Kanten miteinander verbunden. Desweiteren benutzt die Spezifikation folgende Label Wildcards:

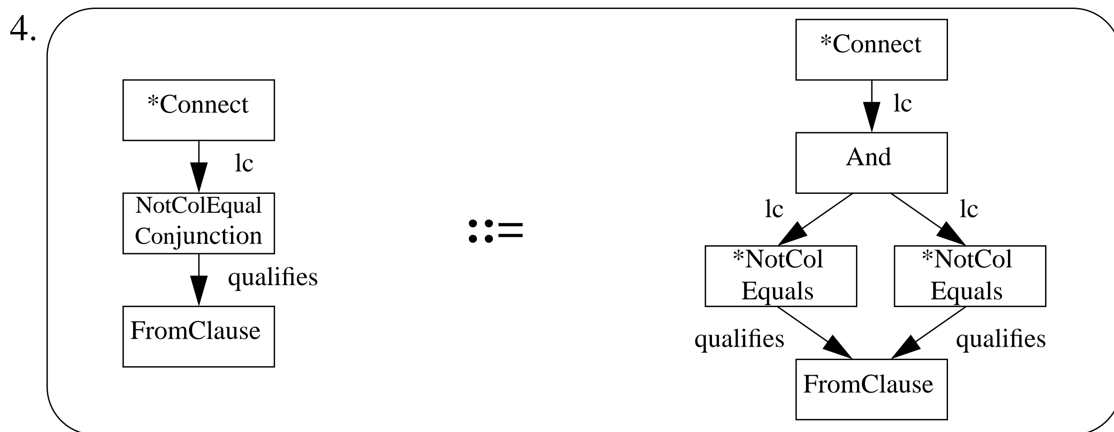
```
*WhereAnd      ∈  {Where, And};
*Cursor        ∈  {Range, Relation};
*NotColEquals  ∈  {NotColEqualConjunction, NotColEqual};
*Connect       ∈  {Or, And, Not, Where};
```



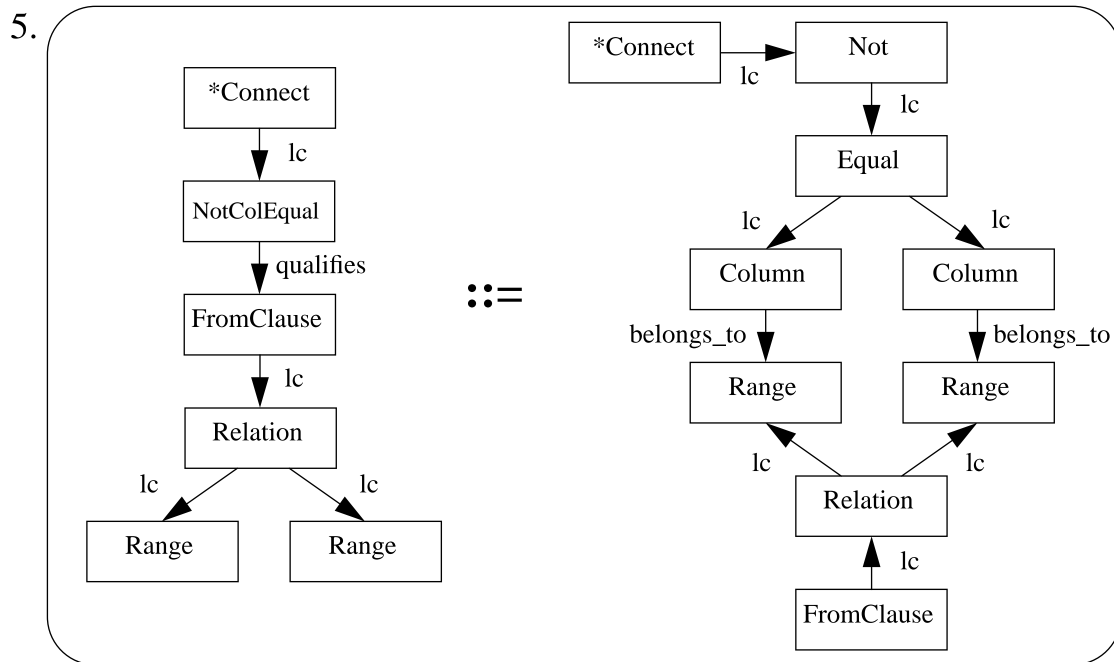
3.



4.



5.



Mit Hilfe der Regel 5 werden alle Konstrukte des abstrakten Syntaxgraphen, die einen Vergleich mit der Bedingung „NOT =“ über gleiche Attributspalten derselben Tabelle enthalten, durch Knoten vom Typ NotColEqual ersetzt und es wird eine qualifies-Kante zur FROM-Klausel des Syntaxbaums der entsprechenden SELECT-Anweisung gezogen. Die Regel 4 ersetzt dann alle Knoten des Typs NotColEqual, die mit Und verknüpft sind, durch einen Knoten vom Typ NotColEqualConjunction. Mit Hilfe der Regel 3 werden diese Knoten dann durch Knoten des Typs NotColEqualConnect ersetzt, der direkt an die WHERE-Klausel gehängt wird. Die genauen Strukturen weiterer angegebener Bedingungen sind dabei nicht von Interesse. Falls der abstrakte Syntaxbaum im Zweig der WHERE-Klausel ausschließlich *eine* Bedingung der Form „NOT =“ über gleiche Attributspalten derselben Tabelle enthält, konnten Regel 3 und 4 hier noch nicht greifen, so daß Regel 2 den NotColEqual-Knoten durch einen Knoten des Typs NotColEqualConnect ersetzt. Nach der Ausführung dieser Regel sind somit alle für ein Cyclic-Exclusion Cliché wichtigen Strukturen erkannt. Dies ist graphisch an einem NotColEqualConnect-Knoten zu erkennen, der direkt über eine lc-Kante mit der WHERE-Klausel und über eine qualifies-Kante mit der FROM-Klausel einer SELECT-Anweisung verbunden ist. Dies wird in Regel 1 dargestellt, die das Cliché erkennt.

Um die korrekte lexikographische Ordnung sicherzustellen, werden die Knoten und Kanten der Produktionen folgendermaßen auf zwei Schichten aufgeteilt:

layer(x) = 0: $\forall(x \in \{\{V\}\backslash NotColEqualConnect\})$

layer(x) = 1: $\forall(x \in \{\{E\}, NotColEqualConnect\})$

Die Überprüfung der lexikographischen Ordnung für die dritte Regel, die Pfadangaben nutzt, kann ohne genauere Berücksichtigung des Pfades erfolgen. Eine Ersetzung des c-Pfades durch lediglich eine lc-Kante wäre hier nämlich der ungünstigste Fall mit $L_0 = 2$ und $R_0 = 3$ für diese Regel.

6.1.3 Group-By Cliché

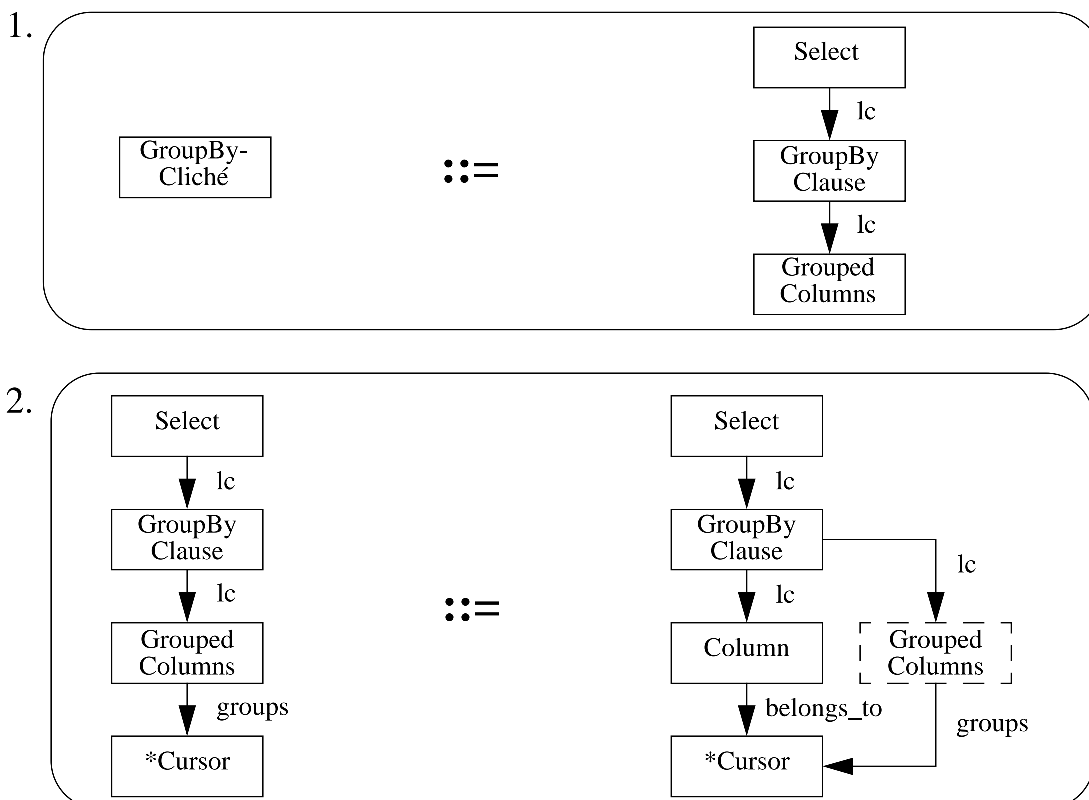
Zur besseren Übersichtlichkeit der Ergebnisrelation wird in Anfragen die GROUP BY-Klausel angegeben. Diese gruppiert die Ergebnisrelation nach den in der Datenbank vorhandenen Ausprägungen der angegebenen Attribute, ohne jedoch eine Sortierung vorzunehmen.

```
select Fachbereich, Name
 from Professor
 group by Fachbereich
```

Die Beispielanfrage sucht den Fachbereich und die Namen aller Professoren und gruppiert diese nach dem Fachbereich. Es ist also relativ schnell ein Überblick darüber zu bekommen, welche Professoren in welchem Fachbereich tätig sind. Die Nutzung einer `GROUP BY`-Klausel macht allerdings nur Sinn, wenn Ausprägungen dieser Attribute, nach denen gruppiert werden sollen, auch mehrfach vorkommen können. Daher scheiden diese Attribute als Schlüsselkandidaten aus.

Die Spezifikation nutzt folgende Label Wildcards:

`*Cursor ∈ {Range, Relation};`



In dieser Spezifikation sucht die Regel 2 zunächst nach einer `GROUP BY`-Klausel in einer `SELECT`-Anweisung. Alle in dieser Klausel aufgeführten Attribute werden mit einem Verweis auf die entsprechende Tabelle durch einen Knoten vom Typ `GroupedColumns` ersetzt. In Regel 1 werden diese Knoten als Group-By Cliché erkannt.

Um die korrekte lexikographische Ordnung sicherzustellen, werden die Knoten und Kanten der Produktionen folgendermaßen auf zwei Schichten aufgeteilt:

layer(x) = 0: $\forall(x \in \{\{V\} \setminus \text{GroupedColumns}\})$

layer(x) = 1: $\forall(x \in \{\{E\}, \text{GroupedColumns}\})$

6.1.4 Complex Cliché

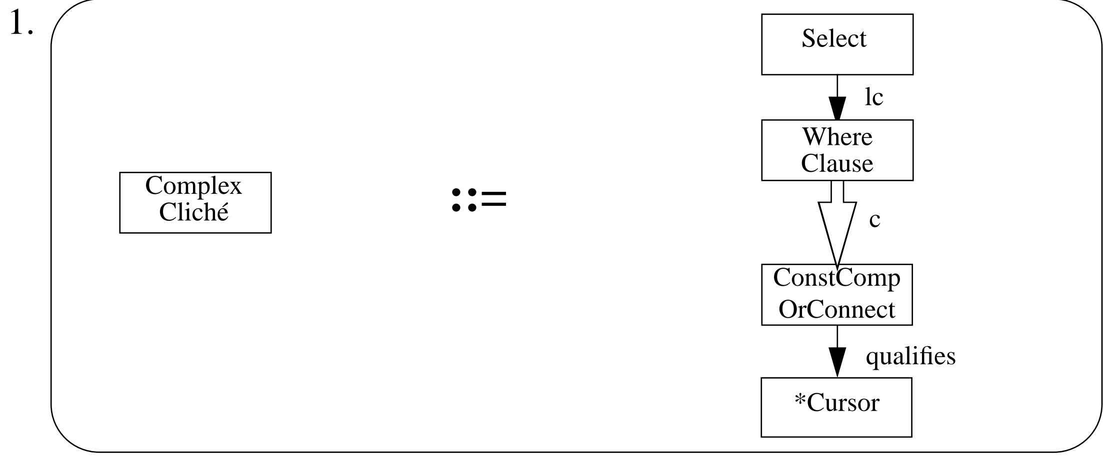
Um Redundanzen und Anomalien zu vermeiden, wird bei der relationalen Datenbanken-bankentwicklung eine Normalisierung vorgenommen (vgl. Kapitel 2.1.3). Dabei verlangt die 1. Normalform, daß alle Attribute einer Relation atomar sind. Um dies zu erreichen, werden deshalb häufig mehrwertige Attribute auf mehrere Spalten einer Relation aufgeteilt. Die Verwendung bestimmter Strukturen in einer SQL-Anweisung kann auf einen solchen Sachverhalt hinweisen.

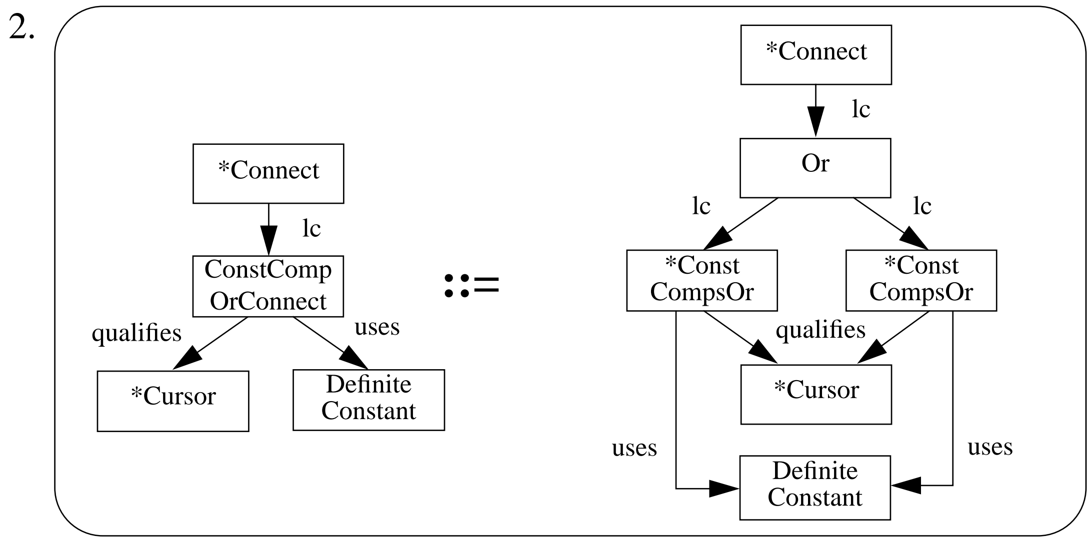
```
select Name
  from Professor
  where Telnr_dienstl = :Tel
        or Telnr_privat = :Tel
```

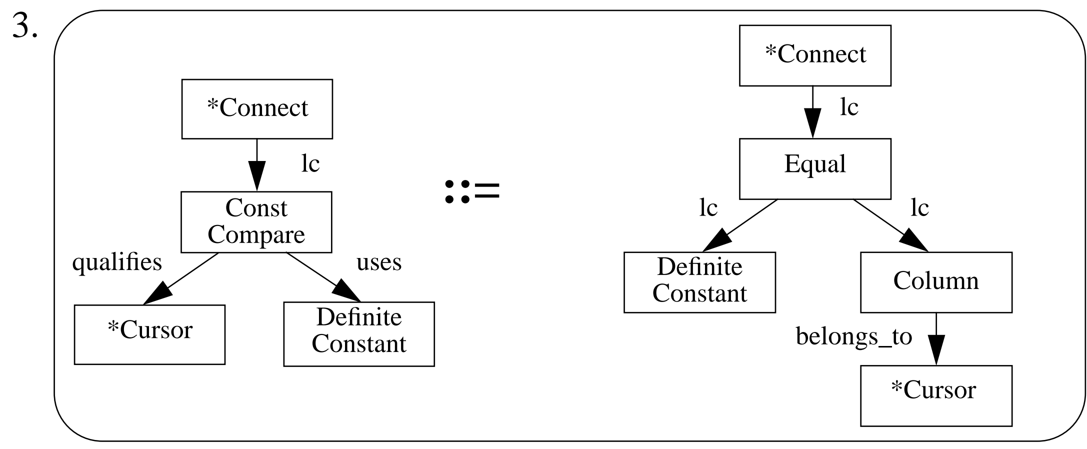
In der vorliegenenden Anweisung wird der Name eines Professors gesucht, der eine bestimmte Telefonnummer hat. Da ein Professor üblicherweise aber sowohl privat als auch dienstlich ein Telefon besitzt, wurden hierfür zwei separate Attribute vorgesehen. Eine Anfrage muß dies berücksichtigen, indem sie, wenn nicht klar ist, welche Telefonnummer vorliegt, beide Attribute aufführt. Eine `WHERE`-Klausel, die eine Bedingung enthält, die eines der beiden Attribute benutzt, muß zum Beispiel auch immer eine `Oder`-Verknüpfung enthalten, die den gleichen Vergleich mit dem anderen Attribut enthält. Es wird also dieselbe Konstante mit unterschiedlichen Attributen einer Relation verglichen.

Die Spezifikation nutzt folgende Label Wildcards:

```
*DefiniteConstant ∈  {Literal, Parameter};
*ConstCompsOr     ∈  {ConstCompOrConnect, ConstCompare};
*Cursor           ∈  {Range, Relation};
*Connect          ∈  {Or, And, Not, Where};
```


1.
Complex
Cliché
::=
Select
lc
Where
Clause
c
ConstComp
OrConnect
qualifies
*Cursor


2.
*Connect
lc
ConstComp
OrConnect
qualifies
uses
*Cursor
Definite
Constant
::=
*Connect
lc
Or
lc
lc
*Const
CompsOr
*Const
CompsOr
qualifies
*Cursor
uses
uses
Definite
Constant


3.
*Connect
lc
Const
Compare
qualifies
uses
*Cursor
Definite
Constant
::=
*Connect
lc
Equal
lc
lc
Definite
Constant
Column
belongs_to
*Cursor

Die Produktion 3 beschreibt die Suche nach einem Vergleich zwischen einem Attribut einer Tabelle und einer Konstanten. Dabei kann eine Konstante nur in Form eines Literals oder Parameters angegeben werden. Alle passenden Strukturen im abstrakten Syntaxgraphen werden mit Produktion 3 durch den Knotentyp `ConstCompare` ersetzt. Alle `Oder`-Verknüpfungen dieser Knoten werden in Regel 2 dann ersetzt durch Konten des Typs `ConstCompOrConnect`. Falls dann ein Knoten dieses Typs gefunden wird, bedeutet dies, daß mindestens 2 Attribute einer Relation mit derselben Konstanten verglichen wurden. Es liegt also ein Complex Cliché vor.

In der oben definierten Spezifikation des Complex Clichés ist in Regel 3 die Überprüfung auf Gleichheit zwischen einem Attribut einer Relaion und einer Konstanten angegeben. Hier könnte jede andere Vergleichsart angenommen werden. Es ist dann allerdings sicherzustellen, daß die Attribute und die Konstante immer mit Hilfe des gleichen Operators verglichen werden. Dies ist bei der Implementierung durch Überprüfung es Knotentyps bzw. entsprechender Attribute zu berücksichtigen.

Um die korrekte lexikographische Ordnung sicherzustellen, werden die Knoten und Kanten der Produktionen folgendermaßen auf zwei Schichten aufgeteilt:
layer(x) = 0: $\forall(x \in \{V\})$
layer(x) = 1: $\forall(x \in \{E\})$

6.2 Clichés zur Erkennung von interrelationalen Abhängigkeiten

6.2.1 Join Cliché

Eine typische Anfrage an eine relationale Datenbank beinhaltet in der `WHERE`-Klausel oft einen Vergleich zwischen Attributen verschiedener Tabellen. Ist dieser Vergleich ein Vergleich auf Gleichheit, kann so ein natürlicher Verbund realisiert werden, der in älteren SQL-Normen nicht explizit in der `FROM`-Klausel angegeben werden konnte.

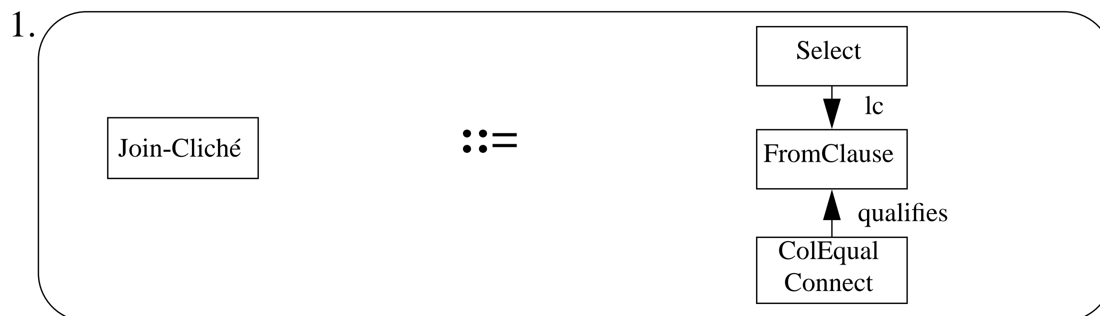
```
select Name
  from Professor P, Vorlesung V
  where V.Professor = P.Pers-Nr
      and V.Bez = :Vorlesung
      and V.Fachbereich = P.Fachbereich
```

Der Vergleich der Attribute `Professor` und `Fachbereich` der Tabelle `Vorlesung` und `Pers-Nr` und `Fachbereich` der Tabelle `Professor` führt in der Beispielanfrage also dazu, daß ein natürlicher Verbund über diese Attribute durchgeführt wird. Dies weist darauf hin, daß die jeweils verglichenen Attribute die gleiche Bedeutung haben und eine Fremdschlüsselbeziehung vorliegen könnte. Allgemein kann man sagen, daß eine Verbundbedingung über die Gleichheit verschiedener Attribute mehrerer Tabellen, die über `Und` miteinander verknüpft sind, auf eine Fremdschlüsselbeziehung zwischen diesen Attributen und Tabellen hinweist. Falls allerdings, wie im Beispiel angegeben, eine andere Bedingung zwischen diesen Vergleichen steht, kann davon ausgegangen werden, daß die verglichenen Attribute nicht zusammen gehören. Im Beispiel ist nur `Pers-Nr` der Schlüssel der Relation `Professor`.

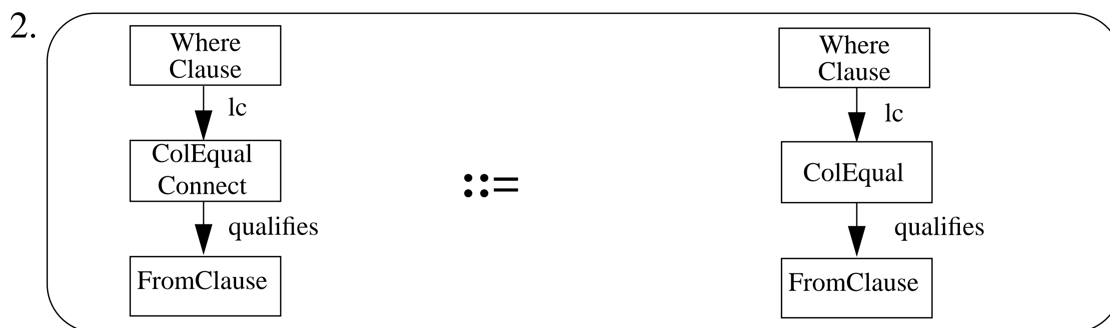
Die Spezifikation nutzt folgende Label Wildcards:

```
*WhereAnd   ∈ {WhereClause, And};
*Cursor     ∈ {Range, Relation};
*ColEquals  ∈ {ColEqualConjunction, ColEqual};
*Connect    ∈ {Or, And, Not, WhereClause};
```

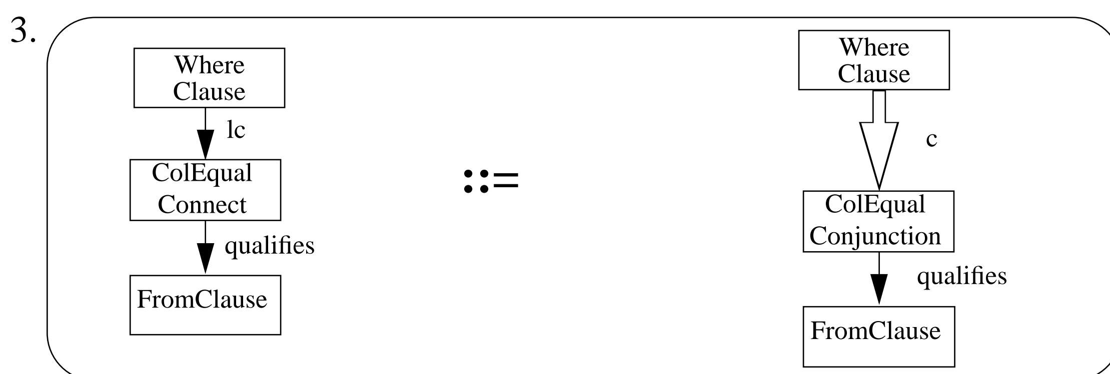
1.



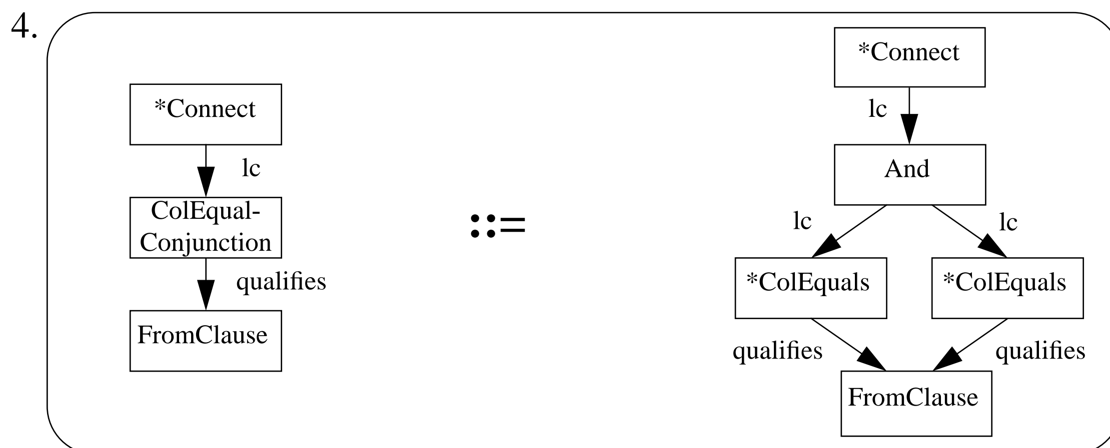
2.

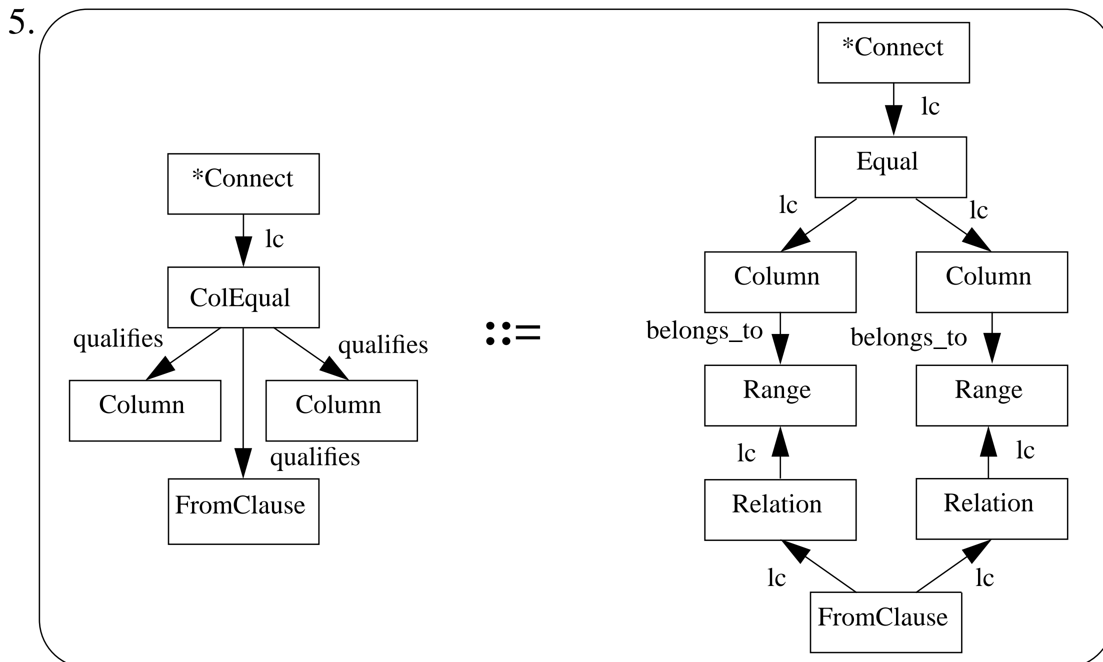


3.



4.





Die Produktion 5 sucht Vergleiche über die Gleichheit zweier Attribute aus unterschiedlichen Tabellen. Dazu werden Gleichheitsbedingungen zwischen Attributen unterschiedlicher Tabellen gesucht, die in der gleichen `FROM`-Klausel angegeben sind. Diese werden durch den Knotentyp `ColEqual` ersetzt. Da die Angabe einer `RANGE`-Variablen nicht zwingend ist, muß dies eventuell durch eine separate Regel für diesen Sachverhalt bei der Implementierung berücksichtigt werden. Die `qualifies`-Kante zu den `Column`-Knoten wird in dieser Spezifikation nicht weiter betrachtet, ist aber bei der Erkennung eines Many-to-Many Clichés wichtig (vgl. Kapitel 6.3.1).

`Und`-Verknüpfungen der mit Hilfe von Produktion 5 erkannten Vergleiche werden mit Regel 4 zu Knoten vom Typ `ColEqualConjuction` zuammengefaßt. Die Regeln 2 und 3 sorgen dafür, daß ein Knoten des Typs `ColEqualConnect` mit Verbindung zur `WHERE`- und zur `FROM`-Klausel einer `SELECT`-Anweisung angelegt wird, falls mindestens eine Gleichheitsbedingung über Attribute verschiedener Tabellen vorliegt. Die Regel 2 spiegelt die Tatsache wider, daß nur ein solcher Vergleich existiert. In Regel 1 wird dann diese Struktur durch einen `Join-Cliché`-Knoten ersetzt.

Um die korrekte lexikographische Ordnung sicherzustellen, werden die Knoten und Kanten der Produktionen folgendermaßen auf zwei Schichten aufgeteilt:

$\text{layer}(x) = 0: \quad \forall (x \in \{\{V\} \setminus \text{ColEqualConnect}\})$

$\text{layer}(x) = 1: \quad \forall (x \in \{\{E\}, \text{ColEqualConnect}\})$

6.3 Clichés zur Erkennung von Optimierungsstrukturen

6.3.1 Many-to-Many Cliché

In Kapitel 2.1.4 wurde am Beispiel beschrieben, wie das Anfrageverhalten einer Datenbank durch Veränderungen im Schema verbessert werden kann. Hier wurde eine N:M-Beziehung zwischen Student und Übung in der Relation `Besuch` abgebildet als:

Besuch:

Übg-Nr	Seq.	Matr-Nr1	Matr-Nr2	....	Matr-Nr10

Eine typische Anfrage, in der diese Tabelle dann angesprochen wird, hat dann diese Struktur:

```
select s.Name
  from Student s, Besuch b, Übung u
  where u.Thema = :T
    and u.Übg-Nr = b.Übg-Nr
    and ( b.Matr-Nr1 = s.Matr-Nr or
          b.Matr-Nr2 = s.Matr-Nr or
          ...
          b.Matr-Nr10 = s.Matr-Nr)
```

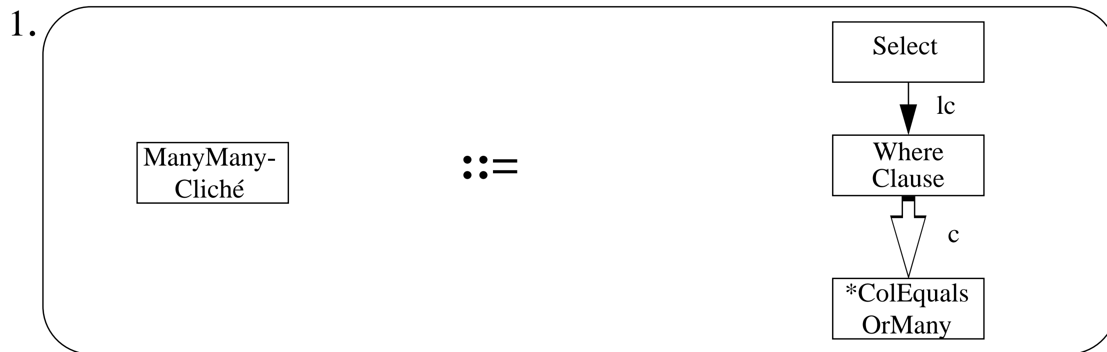
Das Ergebnis dieser Anfrage sind die Namen aller Studenten, die an einer Übung zu einem bestimmten Thema teilnehmen. Dabei weist die Bedingung `u.Übg-Nr = b.Übg-Nr` zunächst einmal darauf hin, daß ein Join-Cliché zwischen `Übung.Übg-Nr` und `Besuch.Übg-Nr` vorliegt. Die nachfolgenden Bedingungen geben einen Hinweis auf eine Fremdschlüsselbeziehung zwischen den Attributen `Besuch.Matr-Nr1`, ..., `Besuch.Matr-Nr10` und `Student.Matr-Nr`. Dabei geht der Vergleich jeweils über das gleiche Attribut der Tabelle `Student`. Diese Art des Vergleiches zeigt an, daß möglicherweise bei der Datenbankentwicklung eine Optimierung vorgenommen wurde und ursprünglich eine N:M-Beziehung vorlag.

Eine solche Anfragestruktur wird im folgenden Many-to-Many Cliché genannt. Dabei ist eine `Oder`-Verknüpfung typisch, die mindestens drei Vergleiche über dasselbe Attribut einer Tabelle mit verschiedenen Attributen einer anderen Tabelle enthält. Diese Forderung nach mindestens drei dieser Vergleiche ist zunächst eine Heuristik, um sicherzugehen, daß eine Optimierungsstruktur vorliegt.

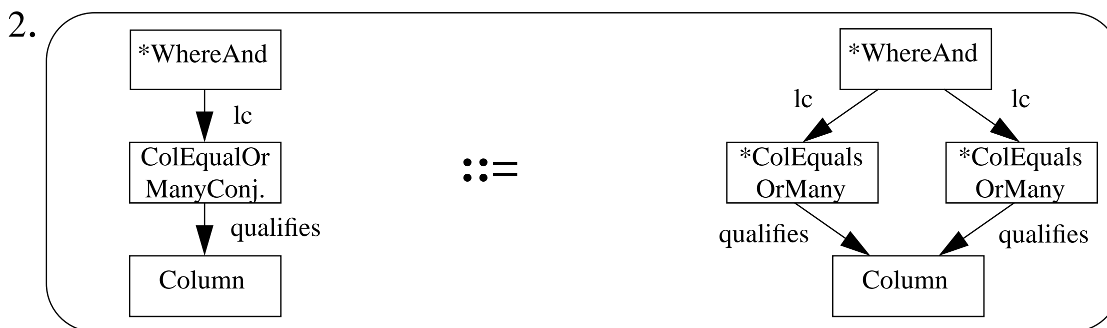
Die Spezifikation nutzt folgende Label Wildcards:

```
*WhereAnd          ∈  {WhereClause, And};
*Cursor            ∈  {Range, Relation};
*ColEqualsOr       ∈  {ColEqualOrConnect, ColEqual};
*ColEqualsOrMany   ∈  {ColEqualOrManyConjunction, ColEqualOrMany};
*Connect           ∈  {Or, And, Not, WhereClause};
```

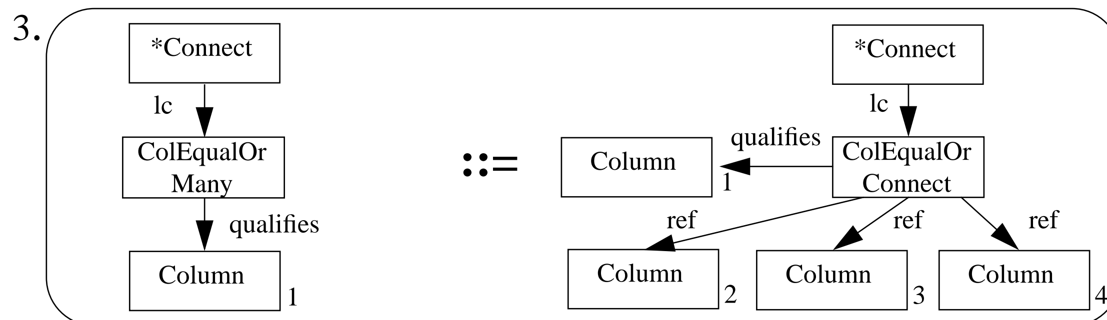
1.

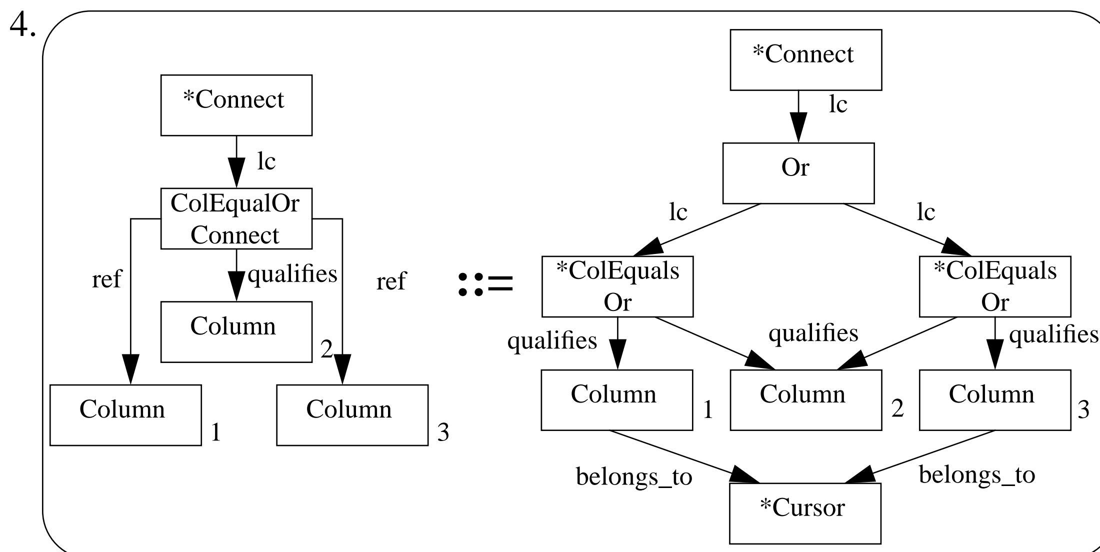


2.



3.



4.


Diese Spezifikation nutzt die Produktion 5 des Join Clichés, die Gleichheitsbedingungen zwischen Attributen unterschiedlicher Tabellen in einer Anweisung heraussucht und durch Knoten vom Typ `ColEqual` ersetzt.

Regel 4 faßt alle `Oder`-Verknüpfungen von Vergleichen eines in allen Verknüpfungen vorkommenden Attributes einer Tabelle mit Attributen anderer Tabellen zusammen zu einem Knoten des Typs `ColEqualOrConnect`. Dabei wird das gemeinsame Attribut durch eine `qualifies`-Kante gekennzeichnet. Die anderen Attribut-Knoten können über `ref`-Kanten erreicht werden.

Die in Regel 4 angelegten `ref`-Kanten werden dann in Regel 3 dazu genutzt, nur die Strukturen weiter zu betrachten, in denen mindestens drei Vergleiche dieser Art enthalten sind. Strukturen dieser Art werden durch Knoten des Typs `ColEqualOrMany` gekennzeichnet. Die Implementierung in Progres benötigt eine Regel in dieser Form nicht, da die Anzahl der Vergleiche hier auch als Attribut eines Knotens gehalten werden kann.

Falls eine N:M-Beziehung, die auf die oben beschriebene Art und Weise optimiert wurde, einen Schlüssel enthält, der sich aus mehreren Attributen zusammensetzt, sind die mit Hilfe von Regel 3 erkannten Strukturen durch `Und` miteinander verknüpft. Diese ersetzt Regel 2 durch Knoten vom Typ `ColEqualOrManyConjunction`. Regel 1 erkennt dann ein Many-to-Many Cliché, falls eine `ColEqualOrMany` oder `ColEqualOrManyConjunction` gefunden wird.

Um die korrekte lexikographische Ordnung sicherzustellen, werden die Knoten und Kanten der Produktionen folgendermaßen auf zwei Schichten aufgeteilt:
layer(x) = 0: $\forall(x \in \{V\})$
layer(x) = 1: $\forall(x \in \{E\})$

6.3.2 Many-to-Many-Select-Distinct Cliché

Über die im letzten Kapitel erwähnte Optimierungsstruktur können noch weitere Informationen durch andere Anfragen gewonnen werden:

select **distinct** s.Name
from Student s, Besuch b
where b.Thema = :T
and (**b.Matr-Nr1 = s.Matr-Nr or**
b.Matr-Nr2 = s.Matr-Nr or
...
b.Matr-Nr10 = s.Matr-Nr)

Diese Anfrage z.B. sucht die Namen der Studenten, die an einer Übung zu einem bestimmten Thema teilgenommen haben. Die Verwendung der `distinct`-Angabe weist dabei darauf hin, daß das Attribut `Name` in der Tabelle `Student` kein Schlüsselattribut ist, da es offensichtlich mehrfach vorkommen kann. Die `WHERE`-Klausel einer solchen Anfrage enthält neben der in Kapitel 6.3.1 beschriebenen `Oder`-Verknüpfung von Vergleichen,verschiedener Attribute einer Tabelle mit jeweils demselben Attribut einer anderen Tabelle nur Bedingungen, die eine Konstante mit einem Attribut vergleichen.

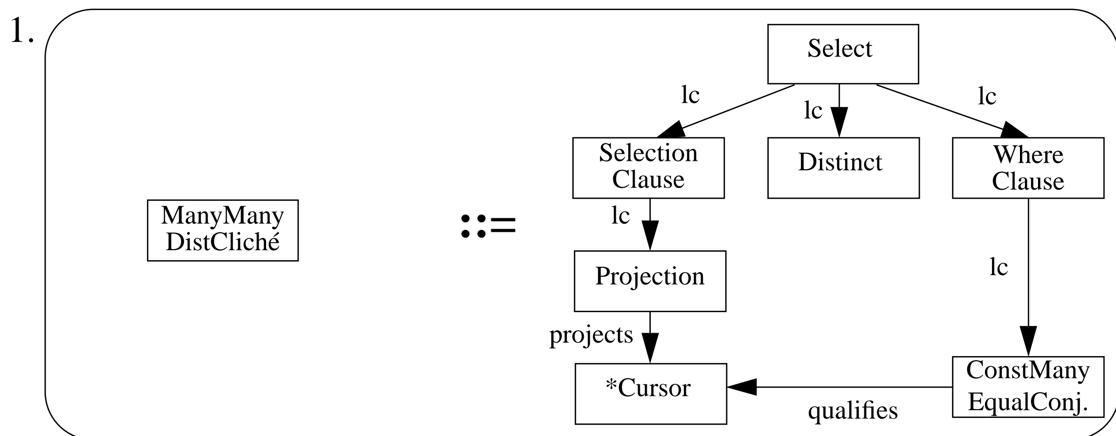
Die Optimierungsstruktur gibt, wie in Kapitel 6.3.1 beschrieben, einen Hinweis darauf, daß das mehrfach genutzte Attribut (hier `s.Matr-Nr`) ein Schlüsselattribut ist, so daß die Rückgaberelation aus einem Tupel besteht. Werden nun weitere Bedingungen angegeben, die mit `Und` verknüpft werden und eine Konstante mit einem Attribut einer Relation vergleichen, weist die Nutzung der `distinct`-Angabe darauf hin, daß hier anlog zum Select-Distinct Cliché kein Schlüsselattribut genutzt wird.

übrige Attribute in Bedingungen, die durch Und verknüpft sind und mit Konstante verglichen werden weisen auf Nicht-Schlüssel hin.

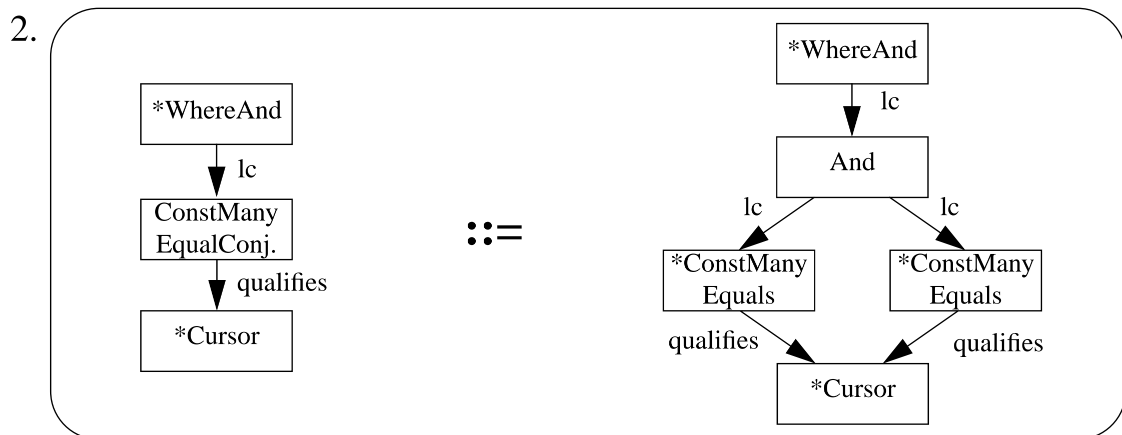
Die Spezifikation nutzt folgende Label Wildcards:

```
*WhereAnd          ∈  {WhereClause, And};
*Cursor            ∈  {Range, Relation};
*ConstEquals[1]    ∈  {ColEqualOrConnect, ConstEqual};
*ColEqualsOrMany   ∈  {ConstEqualConjunction, ColEqualOrMany};
*ConstManyEquals   ∈  {*ColEqualsOrMany, *ConstEquals,
                       ConstManyEqualConjunction};
```

1.



2.



Die Spezifikation dieses Clichés ähnelt der des Many-to-Many Clichés und nutzt die Produktionen 2, 3 und 4 des Many-to-Many Clichés, die Produktion 5 des Join Clichés und die Produktionen Select-Distinct Clichés. Die Produktion 5 des Join Clichés sucht Gleichheitsbedingungen zwischen Attributen unterschiedlicher Tabellen in einer `SELECT`-Anweisung heraus und ersetzt diese Knoten vom Typ `ColEqual`. Regel 4 des Many-to-Many Clichés faßt alle `Oder`-Verknüpfungen von Vergleichen eines in allen Verknüpfungen vorkommenden Attributes einer Tabelle mit Attributen anderer Tabellen

1. vgl Kapitel 6.1.1

zusammen zu einem Knoten des Typs `ColEqualOrConnect`. Dabei wird das gemeinsame Attribut durch eine `qualifies`-Kante gekennzeichnet. Die anderen Attribut-Knoten können über `ref`-Kanten erreicht werden. Diese `ref`-Kanten werden dann in Regel 3 des Many-to-Many Clichés dazu genutzt, nur die Strukturen weiter zu betrachten, in denen mindestens drei Vergleiche dieser Art enthalten sind. Strukturen dieser Art werden durch Knoten des Typs `ColEqualOrMany` gekennzeichnet. `Und`-Verknüpfungen dieser Strukturen werden mit Hilfe von Regel 2 des Many-to-Many Clichés zusammengefaßt zu Knoten des Typs `ColEqualOrManyConjunction`.

Auch in dieser Spezifikation wird zunächst eine Optimierungsstruktur erkannt, bei der das gemeinsam genutzte Attribut auf einen Schlüsselkandidaten hinweist. Diese Strukturen können untereinander durch `Und` verknüpft sein, wie es bei der Spezifikation des Many-to-Many Clichés beschrieben wurde. Hier ist zusätzlich eine `Und`-Verknüpfung mit den in Kapitel 6.1.1 genutzten `ConstEquals` möglich. Die alle Und-Verknüpfungen von Vergleichen zwischen einer Konstanten und Attributen einer Relation zusammenfassen. Regel 2 ersetzt diese Strukturen durch einen Knoten des Typs `ConstManyEqualConjunction`.

7 Zusammenfassung und Ausblick

Ein Datenbank Reverse Engineering-Prozeß beginnt mit der Suche nach semantischen Informationen in den persistenten Strukturen der Datenbank. Dabei bieten die Zugriffe auf die relationalen Datenbanken ein breites Spektrum an implizit enthaltenen semantischen Informationen, wie z.B. referentielle Integritätsbedingungen, Schlüsselkandidaten, Vererbungs- und Aggregationsbeziehungen. Diese Informationen sind oft nicht explizit im Schema der Datenbank enthalten, sondern können erst durch das Auffinden von Hinweisen in Form von typischen Anfragemustern offengelegt werden.

Diese Arbeit stellt einen Ansatz zur Erkennung dieser Clichés im Bereich relationaler Datenbankanwendungen vor. Es wurde prototypisch ein Clichéparser implementiert, der in der VARLET Datenbank Reengineering-Umgebung zur Cliché-Erkennung eingesetzt werden soll und somit die Grundlage für weitere semantische Analysen ist. Da bislang noch keine Cliché-Erkennung existiert, die sich speziell mit der semantischen Analyse von relationalen Datenbankanwendungen beschäftigt, war es auch Aufgabe dieser Arbeit, typische Muster in Datenbankanfragen formal zu spezifizieren. Dies erfolgte mit Hilfe von Layered Graphgrammatiken, die die Spezifikation komplexer Sachverhalte auf dem hohen Abstraktionsniveau von graphischen Beschreibungssprachen erlauben. Da bei dieser Klasse von Graphgrammatiken die linke Seite einer Produktion lexikographisch kleiner als die rechte Seite ist, wird zudem die korrekte Terminierung eines Parsingalgorithmus garantiert.

Die Vorgehensweise des hier entwickelten Clichéparsers orientiert sich am Ablauf existierender Cliché-Erkennungswerkzeuge. Dabei werden die Anfragen an die Datenbank, die standardmäßig als SQL-Anfragen formuliert sind, in einen abstrakten Syntaxgraphen überführt, der dann nach Clichés durchsucht wird, die in einer Clichébibliothek spezifiziert sind.

Die in dieser Arbeit vorgestellten und vom realisierten Clichéparser erkannten Clichés erheben keinen Anspruch auf Vollständigkeit. Es sind vielmehr noch weitere Clichés denkbar. So gibt es z.B. in vielen Unternehmen Programmierrichtlinen, die Spezifikationen von Clichés zulassen, die nur für dieses Unternehmen anwendbar sind. Eine Erweiterung der Clichémenge im Hinblick auf neue SQL-Normen, an denen zur Zeit gearbeitet wird, ist auch denkbar. Diese Arbeit stellt allerdings ein Rahmenwerk zur Verfügung, das es ermöglicht, die Clichébibliothek für Clichés im Bereich relationaler

Datenbankanwendungen auf einfache Art zu erweitern. Es bietet eine Möglichkeit, Clichés formal zu spezifizieren und ermöglicht eine einfache Implementierung auf Basis der Spezifikation.

Der hier entwickelte Rahmen zur Cliché-Erkennung beschränkt sich bisher auf die Betrachtung von Embedded SQL-Anweisungen. Die Erweiterung des entwickelten SQL-Parsers bzw. die Neuentwicklung eines speziellen Parsers würde auch die Analyse von speziellen SQL-Dialekten wie COBOL/SQL oder Anwendungen, die Dynamic SQL verwenden, erlauben.

Es sollte aber nicht vergessen werden, daß Clichés keine unumstößlichen Fakten sind. Es wurden Beispiele genannt, in denen zwei Clichés zu gegensätzlichen Ergebnissen kommen. Die Überprüfung erkannter Clichés durch einen Experten bzw. durch weitere Analysewerkzeuge ist also notwendig. In [HEI98] und [JSZ97] wird die weitere Analyse in VARLET auf Basis unscharfer Logik näher beschrieben.

Diese unscharfe Logik könnte auch schon in der Cliché-Erkennung Anwendung finden, um die Effizienz des vorgestellten Clichéerkenners zu steigern. Dazu könnten Abhängigkeiten zwischen den einzelnen Clichés als *Generic Fuzzy Reasoning Nets* (GFRN) [JSZ97] formuliert werden, mit denen sich Sachverhalte, die auf unscharfer Logik beruhen, beschreiben lassen. Bei der Cliché-Erkennung könnte beschrieben werden, welche Clichés nach dem Auffinden bestimmter Konstrukte hieraus noch resultieren können. Beispielsweise sollte nach einem Many-to-Many Cliché nur weiter gesucht werden, wenn mindestens drei Vergleiche über das gleiche Attribut gefunden wurden (vgl. Kapitel 6.3.1). Dies wäre, ähnlich wie es in [QUI94] beschrieben wird, eine Möglichkeit, die Anzahl der betrachteten Clichés einzuschränken. Allerdings wäre mit den GFRN eine graphische Spezifikation dieser Abhängigkeiten auf abstrakter Ebene möglich, so daß eine aufwendige manuelle Spezifikation der möglichen Einschränkungen, wie sie in [QUI94] notwendig ist, entfällt.

8 Anhang

A SQL-DML Grammatik

In diesem Anhang wird eine Beschreibung der SQL-Syntax in Backus-Naur-Form angegeben, die die für die Cliché-Erkennung wichtigen DML-Anweisungen der Structured Query Language (SQL) umfaßt (vgl. [DAT89], [LMB92]). Sie ist Teil der Grammatik, die der Clichéparser verwendet. Dabei ist `sql` das Startsymbol, aus dem sowohl `manipulative_statement` wie auch `query_exp` (vgl. Seite 93) abgeleitet werden. Die großgeschriebenen Bezeichner weisen die Terminalsymbole aus, während die kleingeschriebenen Bezeichner auf Nichtterminalsymbole hinweisen.

```
sql:
    manipulative_statement ;
manipulative_statement:
    delete_statement_positioned
  | delete_statement_searched
  | insert_statement
  | select_statement
  | update_statement_positioned
  | update_statement_searched
  ;
delete_statement_positioned:
    DELETE FROM table WHERE CURRENT OF cursor ;
delete_statement_searched:
    DELETE FROM table opt_where_clause ;
insert_statement:
    INSERT INTO table opt_column_commalist
    values_or_query_spec
  ;
values_or_query_spec:
    VALUES '(' insert_atom_commalist ')'
  | query_spec
  ;
```

```
insert_atom_commalist:
    insert_atom
  | insert_atom_commalist ',' insert_atom
  ;

insert_atom:
    atom
  | NULLX
  ;

select_statement:
    SELECT opt_all_distinct selection
    INTO target_commalist
    table_exp
   ;

opt_all_distinct:
    /* empty */
  | ALL
  | DISTINCT
  ;

update_statement_positioned:
    UPDATE table SET assignment_commalist
    WHERE CURRENT OF cursor
  ;

assignment_commalist:
  | assignment
  | assignment_commalist ',' assignment
  ;

assignment:
    column EQUAL scalar_exp
  | column EQUAL NULLX
  ;

update_statement_searched:
    UPDATE table SET assignment_commalist opt_where_clause
  ;

target_commalist:
    target
  | target_commalist ',' target
  ;

target:
    parameter_ref ;

opt_where_clause:
    /* empty */
  | where_clause
;
```

```
sql:
   query_exp ;

query_exp:
   query_term
 | query_exp UNION query_term
 | query_exp UNION ALL query_term
 ;

query_term:
   query_spec
 | '(' query_exp ')'
 ;

query_spec:
   SELECT opt_all_distinct selection table_exp
 ;

selection:
   scalar_exp_commalist
 | '*'
 ;

table_exp:
   from_clause
   opt_where_clause
   opt_group_by_clause
   opt_having_clause
 ;

from_clause:
   FROM table_ref_commalist ;

table_ref_commalist:
   table_ref
 | table_ref_commalist ',' table_ref
 ;

table_ref:
   table
 | table range_variable
 ;

where_clause:
   WHERE search_condition ;

opt_group_by_clause:
   /* empty */
 | GROUP BY column_ref_commalist
 ;
```

```
column_ref_commalist:
    column_ref
  | column_ref_commalist ',' column_ref
  ;

opt_having_clause:
    /* empty */
  | HAVING search_condition
  ;

search_condition:
  | search_condition OR search_condition
  | search_condition AND search_condition
  | NOT search_condition
  | '(' search_condition ')'
  | predicate
  ;

predicate:
    comparison_predicate
  | between_predicate
  | like_predicate
  | test_for_null
  | in_predicate
  | all_or_any_predicate
  | existence_test
  ;

comparison_predicate:
    scalar_exp COMPARISON scalar_exp
  | scalar_exp COMPARISON subquery
  ;

between_predicate:
    scalar_exp NOT BETWEEN scalar_exp AND scalar_exp
  | scalar_exp BETWEEN scalar_exp AND scalar_exp
  ;

like_predicate:
    scalar_exp NOT LIKE atom opt_escape
  | scalar_exp LIKE atom opt_escape
  ;

opt_escape:
    /* empty */
  | ESCAPE atom
  ;
```

```
test_for_null:
     column_ref IS NOT NULLX
    |column_ref IS NULLX
  ;
in_predicate:
    scalar_exp NOT IN  subquery
  | scalar_exp IN  subquery
  | scalar_exp NOT IN '(' atom_commalist ')'
  | scalar_exp IN '(' atom_commalist ')'
  ;
atom_commalist:
    atom
  | atom_commalist ',' atom
  ;
all_or_any_predicate:
    scalar_exp COMPARISON any_all_some subquery ;
any_all_some:
    ANY
  | ALL
  | SOME
  ;
existence_test:
    EXISTS subquery ;
subquery:
    '(' SELECT opt_all_distinct selection table_exp ')'
;
scalar_exp:
    scalar_exp '+' scalar_exp
  | scalar_exp '-' scalar_exp
  | scalar_exp '*' scalar_exp
  | scalar_exp '/' scalar_exp
  | '+' scalar_exp %prec UMINUS
  | '-' scalar_exp %prec UMINUS
  | column_ref
  | function_ref
  | atom
  | '(' scalar_exp ')'
  ;
scalar_exp_commalist:
    scalar_exp
  | scalar_exp_commalist ',' scalar_exp
  ;
```

```
atom:
    parameter_ref
  | literal
  | USER
  ;

parameter_ref:
    parameter
  | parameter parameter
  | parameter INDICATOR parameter
  ;

function_ref:
    AMMSC '(' '*' ')'
  | AMMSC '(' DISTINCT column_ref ')'
  | AMMSC '(' ALL scalar_exp ')'
  | AMMSC '(' scalar_exp ')'
  ;

literal:
    STRING
  | INTNUM
  | APPROXNUM
  ;

table:
    NAME
  | USER
  | ORDER
  | NAME '.' NAME
  ;

column_ref:
    NAME
  | NAME '.' NAME
  | NAME '.' NAME '.' NAME
  ;

column:
    NAME
  | EXECUTE
  ;

parameter:
    ':' NAME ;

range_variable:
    NAME ;
```

Literatur

[AND94] Andersson, M.: *Extracting an Entity Relationship Schema from a Relational Database through Reverse Engineering.* Proceedings of the 13th Conference of ERA, Manchester 1994.

[BEN90] Bennett, J.P.: *Introduction to Compiling Techniques: A First Course using ANSI C, LEX and YACC.* McGRAW-HILL Book Comany (UK) Ltd, 1990.

[BER90] Bertelsmann Universal Lexikon, Bertelsmann Lexikon Verlag GmbH, 1990.

[DAT89] Date, C.J.: *A Guide to the SQL Standard, 2nd ed.* Addison–Wesley Publishing Company, 1989.

[EN94] Elmasri; Navathe: *Fundamentals of Database Systems, 2nd ed.* Benjamin/ Cummings Publishing Company, Inc, 1994.

[FV95] Fahrner; Vossen: *Transforming Relational Database Schemas into Object-Oriented Schemas according to ODMG93.* Proceedings of the 4th International Conference on Deductive and Object-Oriented Databases, 1995.

[HEH+98] Henrard; Englebert; Hick; Roland; Hainaut: *Program understanding in databases reverse engineering.* Research Paper RP-98-004, Universität Namour, 1998.

[HEI98] Heitbreder, Melanie: *Eine Ausführungsmaschine für Generic Fuzzy Reasoning Nets auf Basis unscharfer Petrinetze.* Diplomarbeit, Universität-Gesamthochschule Paderborn, 1998.

[HER92] Herold, Helmut: *Lex und Yacc: lexikalische und syntaktische Analyse.* Addison-Wesley, 1992.

[HS95] Heuer; Saake: *Datenbanken: Konzepte und Sprachen.* International Thomson Publishing GmbH, 1995.

[HTJ+93] Hainaut; Tonneau; Joris; Chandelon: *Schema transformation techniques for database reverse engineering.* Research Paper RP-93-004, 12th Conference on Entity-Relationship Approach, Arlington, 1993

[JSZ97] Jahnke, J. H.; Schäfer, W.; and Zündorf, A.: *Generic fuzzy reasoning nets as a basis for reverse engineering relational database applications*. In *Proc. of European Software Engineering Conference (ESEC/FSE)*, number 1302 in LNCS. Springer, September 1997.

[JZ98] Jahnke, Jens H.; Zündorf, Albert: *Using Graph Grammars for Building the Varlet Database Reverse Engineering Environment.* Angenommen für Proceedings 6th International Workshop on Theory and Application of Graph Transformations, TAGT‘98, Paderborn, 1998

[LMB92] Levine; Mason; Brown: *lex & yacc.* O’Reilly & Associates Inc, 1992.

[NAG79] Nagl, Manfred: *Graph-Grammatiken: Theorie, Anwendungen, Implementierung.* Vieweg-Verlag, 1979.

[PB94] Premerlani; Blaha: *An Approach for Reverse Engineering of Relational Databases.* Communications of the ACM, Volume 37(5), S. 42 - 49,1994.

[PB95] Premerlani; Blaha: *Observed Idiosyncracies of Relational Database Design.* 2nd Working Conference on Reverse Engineering, Toronto,1995.

[PP94] Paul; Prakash: *A Framework for Source code Search using Program Patterns.* Aus: IEEE Transactions on Software Engineering, Vol 20, No.6, 6/94.

[QUI94] Quilici, Alex: *An Opportunistic, Memory-Based Approach to Recognizing Programming Plans,* Communications of the ACM, Volume 37(3), S. 84 - 95, 1994.

[RS95] Rekers; Schürr: *A parsing algorithm for context-sensitive graph-grammars (long version).* Technical Report 95-05, Universität Leiden, Niederlande, 1995.

[RS97] Rekers; Schürr: *Defining and Parsing Visual Languages with Layered Graph Grammars.* Journal of Visual Languages and Computing, Vol 8, London, Academic Press, 1997

[RW90] Rich; Wills: *Recognizing a Program's Design: A Graph-Parsing Approach.* Aus: Arnold, Robert S.: *Software Engineering*. IEEE Computer Society Press, 1990.

[ROS97] Rosenberg, Grzegorz: *Handbook of Graph Grammars and Computing by Graph Transformation*. World Scientific, Singapore, 1997.

[SCH87] Schütte, Alois: *Spezifikation und Generierung von Übersetzern für Graph-Sprachen durch attributierte Graph-Grammatiken,* EXpress Edition GmbH, 1987.

[SCH95] Schäfer, Wilhelm: *Skript zur Vorlesung: Datenbanken und Informationssysteme*, Universität-Gesamthochschule Paderborn, 1995.

[SCH96a] Schürr, Andy: *PROGRES for Beginners*, Technical Report, Lehrstuhl für Informatik III RWTH Aachen, 1996.

[SCH96b] Schürr, Andy: *A Guided Tour Through the PROGRES Environment*, Technical Report, Lehrstuhl für Informatik III RWTH Aachen, 1996.

[SIM95] Simon, Alan R.: *Strategic Database Technology: Management for the Year 2000,* Morgan Kaufmann Publishers, Inc., 1995.

[STA89] Stahlknecht, Peter: *Einführung in die Wirtschaftsinformatik,* 4. Auflage, Springer-Verlag, 1989.